Practical Aspects of Probabilistic Model Checking

Ernst Moritz Hahn and *Andrea Turrini*

State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences

`http://iscasmc.ios.ac.cn`

July 30, 2016

Software systems are ubiquitous

Software systems are ubiquitous:

- smartphones and laptops
- medical systems
- communication systems
- transportation systems

and they are getting more involved day by day.

When things go wrong

System: an app on the phone
Problem: it works badly
Effect: depending on the app, you can

- reach a different place
- run out of battery quicker than expected

Cost: probably nothing, or negligible

Not too problematic, just look for an update/different app

When things go wrong

System: program on a laptop
Problem: unexpected termination
Effect: depending on the program and your usual behaviour,

- you lose some work
- you have to start again

Cost: probably negligible

Not too problematic, just do again the lost work... and learn to save frequently

When things go really wrong

System: Therac-25, a radiation machine for cancer treatment
Problem: design error in the control software
Effect: huge over-dosis ($\approx$ x100) of the radiation
Cost: three patients died

When things go really wrong

System: AT&T telephone system
Problem: software flaw (in the meaning of a `break` statement)
Effect: 9h outage in a large part of US phone system
Cost: several 100M US$

When things go really wrong

System: Ariane-5 heavy lift launch rocket
Problem: software flaw in the control software
- caused by a conversion of a 64-bit floating point to 16-bit signed integer
- this would have been managed by the corresponding software handler, but it was disabled for efficiency

Effect: rocket exploded
Cost: more than 500M US$

When things go really wrong

System: Flight AF-447 Rio de Janeiro–Paris
Problem: Several causes, including
- wrong speed indication
- incomplete information to the pilots
- untrained pilots about autopilot disabling in uncommon situations

Effect: Plane crashed in the ocean
Cost: More than 200 casualties

Why verification is important

Why verification?

Systems can go wrong since they may contain bugs.
This can cost a lot of money and even lifes.

Note that we have to be sure about what we want to check:

- check that we are building the **thing right**: *verification*
  Therac-25, AT&T, Ariane-5
- check that we are building the **right thing**: *validation*
  AF-447

How to verify

There are several ways to verify a system:

Peer reviewing

- manual inspection of the system by a colleague
- static approach
- common programs contain millions of lines of code
- it is not feasible when the system gets larger
- subtle errors usually hidden in components working in parallel
- subtle errors hard to be discovered
- error detection rate: between 30% and 90%, median 60%

How to verify

There are several ways to verify a system:

Simulation and testing

- the system is executed on different inputs to see whether errors occur
- it remains feasible even when the system gets larger
- it can be extended to systems expecting user interaction, by synthesising the user behaviour
- it can be automated
- when an error occurs, the system is buggy
- what about when no error occurs?

The importance of formal verification

If we want to be sure that a system behaves as expected, we have to adopt techniques guaranteeing the validity of the response.

Be formal

Formal verification techniques provide such a guarantee.

Formal verification methods

Given a property the system has to satisfy,

Deductive methods

- method: generate a formal proof that the system satisfies the property
- tool: theorem prover, proof assistant/checker (Coq, Isabelle, ...)
- applicable if: the system can be represented as a mathematical theory

Model checking methods

- method: systematic check that the property holds in the system
- tool: model checker (NuSMV, Spin, UppAal, ...)
- applicable if: the system generates a (finite) behavioural model

Simulation methods

- method: check the property by exploring all behaviours of the system
- applicable if: the system defines an executable model

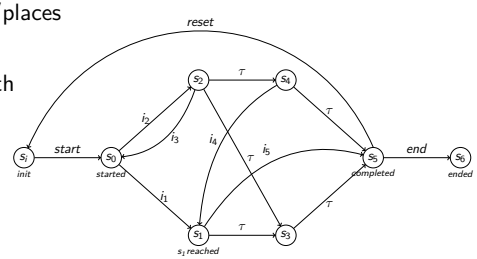
Behavioural model, informal description

A behavioural model represents how a system can behave.
It is usually represented as

- a set of states/locations/places
- a set of transitions

A model can be decorated with

- labels on transitions
- labels on states

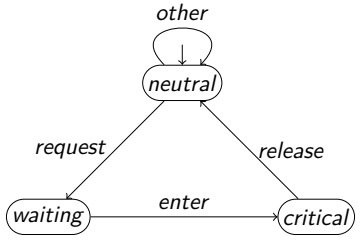


Behavioural model, a real example

As real system, consider the mutual exclusion problem:

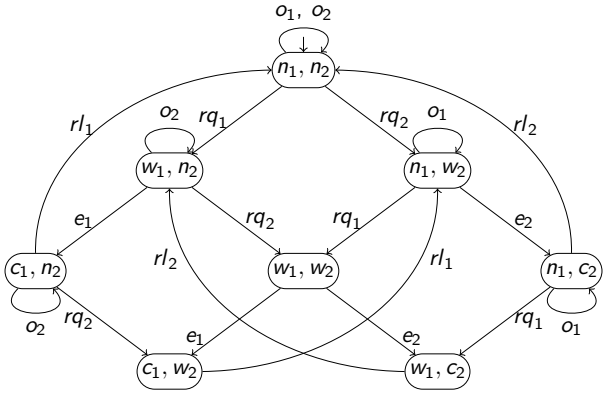
- there are n processes that have to access a shared resource
- only one process at a time can access the shared resource
- accessing the shared resource corresponds to enter a 1-slot critical section

For the case $n = 1$ process, we can represent it as the system



Behavioural model, a real example

For the case $n = 2$ processes, we can represent it as the system

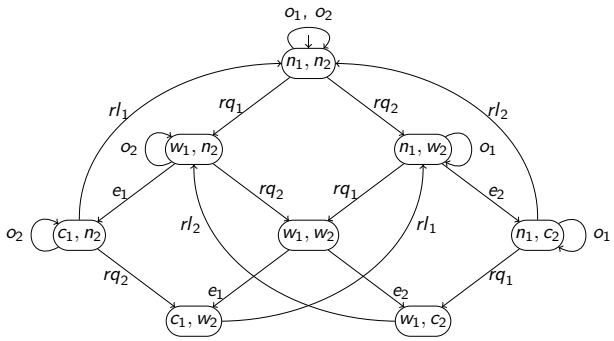


Why model checking is important

Given a model, we want to be sure that it satisfies the given properties.
Some property is easy to check:

Easy property

It never happens that two processes are in the critical section at the same time.

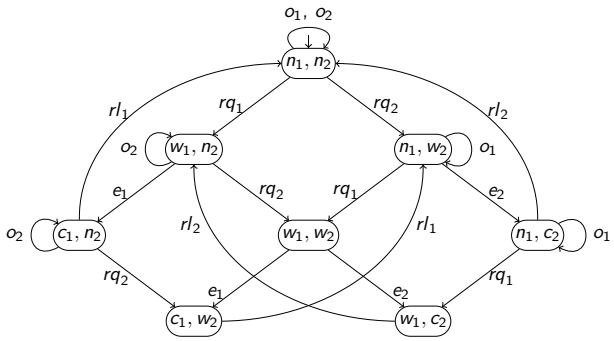


Why model checking is important

Given a model, we want to be sure that it satisfies the given properties.
Some other property is not so easy to check:

Not so easy property

Whenever a process wants to enter the critical section, it will eventually enter.



Why model checking is important

Model checking, the idea

Model checking is an automatic technique that, given a finite state model of the system and a formal property, systematically checks whether such a property holds for (a given state in) that model.

How model checking works

Model checking is the result of multiple steps:

1. modelling phase:
   - create a model for the system to analyze
   - formalize the property to check
2. check phase:
   - run the model checker on the model and property
3. result analysis phase:
   - if the property is satisfied, that's all
   - if the property is not satisfied,
     1. get a counterexample and study it
     2. refine the model, the property, or the system, and try again
   - if no answer obtained, try to reduce the size of the model so to reduce the requirements for the model checker

The pros of model checking

- widely applicable: hardware, software, protocols, ...
- allows for partial verification: check only interesting properties
- potential "push-button" technology: software tools
- rapidly increasing industrial interest
- counterexamples for property violation
- sound and interesting mathematically foundations
- not biased to the most possible scenarios

The cons of model checking

- main focus on control-intensive application, less on data-oriented
- model checking ensures the result on the model, not on the modelled system
- no guarantee about completeness of the results
- impossible to check generalizations

Why model checking is important

Model checking is a very effective technique to analyze systems and expose design errors

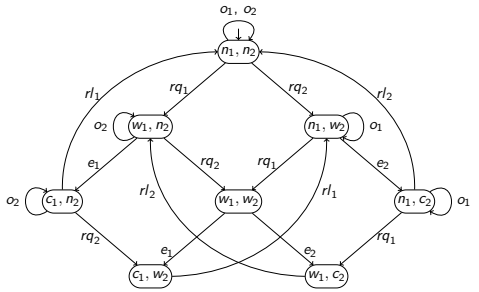
Real world usages:

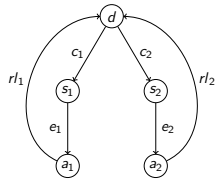
- security: Needham-Schroeder encryption protocol: flaw discovered after 17 years
- transportation systems: train model with 10^{476} states
- analysis of C, C++ and Java programs:
  - model checker used and developed by Microsoft, Digital, NASA
  - successful analysis of device drivers
- software in aerospace industry: NASA's Mars Pathfinder, Deep Space-1, JPL LaRS group

Again on the mutual exclusion problem

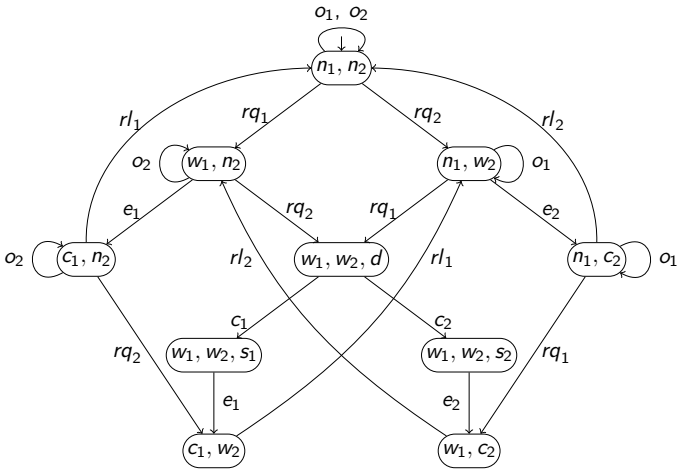
Not so easy property

Whenever a process wants to enter the critical section, it will eventually enter.

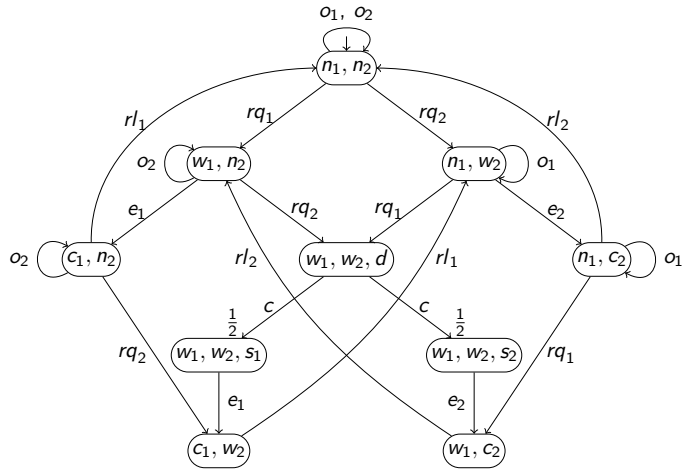




The resulting model for the mutual exclusion problem



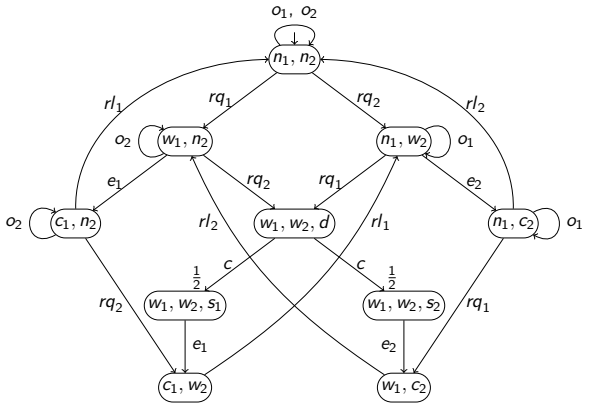
A probabilistic mutual exclusion model



A probabilistic mutual exclusion

A new type of properties

With probability 1,
whenever a process wants to enter the critical section, it will eventually enter.



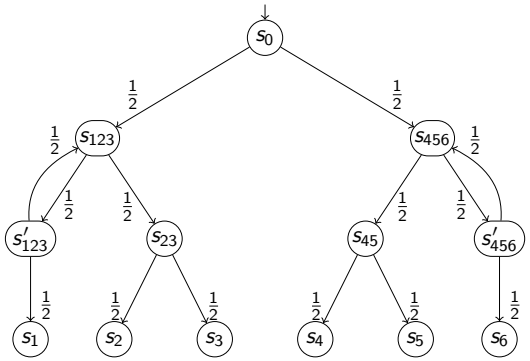
Why probability is important

Probability is the core part for several systems and situations:

- randomized algorithms
- complexity reduction
- probabilistic programming
- reliability
- performance
- optimization
- system biology

Why probability is important: randomized algorithms

Simulating a fair dice with a fair coin:



Is it really a fair dice?

Why probability is important: distributed computing

FLP impossibility result [Fischer *et al.*, 1985]

In an asynchronous setting, where only one processor might crash, there is **no** distributed algorithm that solves the consensus problem: getting a distributed network of processors to agree on a common value.

Ben-Or's possibility result [Ben-Or, 1983]

If a process can make a decision based on its internal state, the message state, and some **probabilistic** state, the consensus in the asynchronous setting can be reached with probability 1.

Why probability is important: complexity reduction

Computing matrix multiplication is time consuming; consider the problem of checking whether $A \times B = C$ for three matrices of size N^2.

Deterministic approach

Algorithm: compute $A \times B$ and compare with C

Complexity: in $\mathcal{O}(N^3)$, with best known complexity in $\mathcal{O}(N^{2.37})$

Probabilistic approach

Algorithm:
1. take a random bit vector $\vec{x} \in \{0,1\}^n$
2. compute $\vec{a} = A \times (B \times \vec{x}) - C \times \vec{x}$
3. if $\vec{a}$ is **not** the null vector, return **no**
4. repeat steps 1-3 for k times, then return **yes**

Complexity: in $\mathcal{O}(k \cdot N^2)$, with error probability at most 2^{-k}

Why probability is important: reliability

Zeroconf protocol objectives

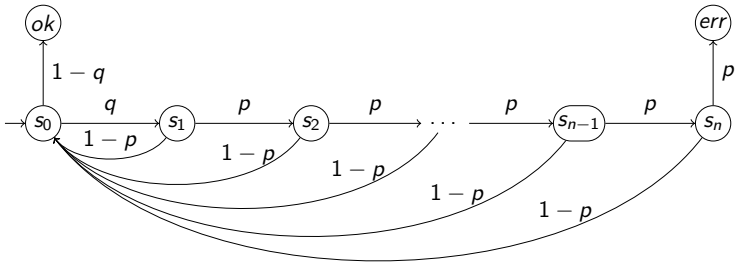
- network protocol for address assignment
- new devices joining the network get a unique IP address
- no user interaction needed

Zeroconf protocol overview

1. randomly choose one of the 65 024 addresses available in the private B-class 169.254.0.0/16
2. Loop: as long as the number of sent probes is less than n
3. broadcast the probe message "who is using the chosen address"?
4. got a reply? Go to 1
5. no reply within $r > 0$ time units:
   - if n probes have been sent: use the address
   - otherwise go to 2

Why probability is important: reliability

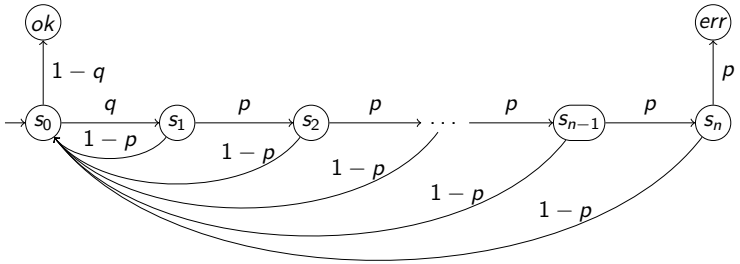
A simplified model for the Zeroconf protocol is:



q: probability of choosing an address already in use, $q = \frac{\#\text{devices}}{65024}$

p: probability of message loss

Why probability is important: reliability



What is the probability that

- an IP address is eventually obtained?
- an unused IP address is eventually obtained?
- an already in use IP address is eventually obtained?

Why probability is important: it helps in several cases

- when modelling and analyzing dependability and reliability
  - to quantify arrivals, message loss, waiting time, ...
- when building protocols for networked embedded systems
  - randomized algorithms, breaking symmetry, ...
- when problems are undecidable
  - like distributed consensus, repeated reachability of lossy channel systems, ...
- for improving performance
  - matrix multiplication, randomized quicksort, ...

Why probabilistic model checking is important

In general,

- complex systems are likely to contain bugs
- finding bugs is important for safety and correctness
- model checking is an effective approach to verify that we are building the things right

Probability, on the other hand,

- can be used for improving performance
- can be mandatory for building the system
- it is not really intuitive
- it is more likely to make mistakes
- it is not easy to verify, but
- some wrong behaviour can be safely ignored

Probabilistic model checking is a very effective technique to analyze **probabilistic** systems and expose design errors

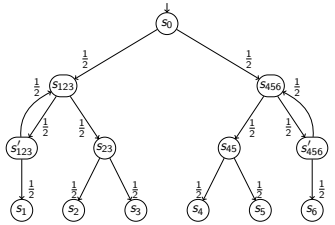
Probabilistic models: Markov chains

A *(Discrete time) Markov chain* (MC) is a tuple $\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$ where

- S is a finite set of states
- $\bar{s}$ is the initial state
- $L\colon S \to \Sigma$ is a labelling function
- $\mathrm{P}\colon S \times S \to [0,1]$ is the transition probability matrix

P is such that $\sum_{s' \in S} \mathrm{P}(s,s') \in \{0,1\}$ for each $s \in S$.

Example of Markov chain

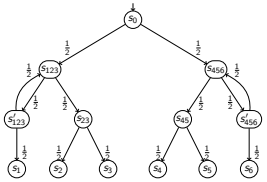


$\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$

- $S = \{s_0, s_{123}, s_{456}, s'_{123}, s_{23}, s_{45}, s'_{456}, s_1, s_2, s_3, s_4, s_5, s_6\}$
- $\bar{s} = s_0$
- $L(s) = s$ for each $s \in S$
- $\mathrm{P}(s_0, s_0) = 0$
  $\mathrm{P}(s_0, s_{123}) = \frac{1}{2}$
  $\mathrm{P}(s_0, s_{456}) = \frac{1}{2}$
  $\ldots$

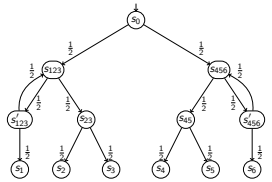
Computing probabilities

What is the probability of finally reaching the state s_2?



$$
\begin{aligned}
&\mathfrak{P}(s_0\ s_{123}\ s_{23}\ \textcircled{s_2}) \\
&+ \mathfrak{P}(s_0\ s_{123}\ s'_{123}\ s_{123}\ s_{23}\ \textcircled{s_2}) \\
&+ \mathfrak{P}(s_0\ s_{123}\ s'_{123}\ s_{123}\ s'_{123}\ s_{123}\ s_{23}\ \textcircled{s_2}) \\
&+ \mathfrak{P}(s_0\ s_{123}\ s'_{123}\ s_{123}\ s'_{123}\ s_{123}\ s'_{123}\ s_{123}\ s_{23}\ \textcircled{s_2}) \\
&+ \mathfrak{P}(s_0\ s_{123}\ s'_{123}\ s_{123}\ s'_{123}\ s_{123}\ s'_{123}\ s_{123}\ s'_{123}\ s_{123}\ s_{23}\ \textcircled{s_2}) \\
&+ \mathfrak{P}(s_0\ s_{123}\ s'_{123}\ s_{123}\ s'_{123}\ s_{123}\ s'_{123}\ s_{123}\ s'_{123}\ s_{123}\ s'_{123}\ s_{123}\ s_{23}\ \textcircled{s_2}) \\
&\ldots \\
&= \sum_{n=0}^{\infty} \mathfrak{P}(s_0\ s_{123}\ (s'_{123}\ s_{123})^n\ s_{23}\ \textcircled{s_2})
\end{aligned}
$$

Computing probabilities



How can we compute $\mathfrak{P}(s_0\ s_{123}\ s_{23}\ \textcircled{s_2})$?

Intuitively, it is

$$\mathfrak{P}(s_0\ s_{123}\ s_{23}\ \textcircled{s_2}) = \mathrm{P}(s_0, s_{123}) \cdot \mathrm{P}(s_{123}, s_{23}) \cdot \mathrm{P}(s_{23}, \textcircled{s_2})$$

Formally, it is

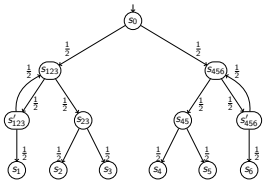
Probability of a path

A finite path ξ is a finite sequence of states $\xi = s_0 s_1 s_2 \ldots s_n$ such that for each $0 \le i < n$, $\mathrm{P}(s_i, s_{i+1}) > 0$.

The probability $\mathfrak{P}(\xi)$ of ξ is defined as $\mathfrak{P}(\xi) = \prod_{i=0}^{n-1} \mathrm{P}(s_i, s_{i+1})$.

Computing probabilities

What is the probability of finally reaching the state s_2?



$$\sum_{n=0}^{\infty} \mathfrak{P}(s_0\, s_{123}\, (s'_{123}\, s_{123})^n\, s_{23}\, (s_2))$$
$$= \sum_{n=0}^{\infty} \frac{1}{2} \cdot \frac{1}{2} \cdot \left(\frac{1}{2} \cdot \frac{1}{2}\right)^n \cdot \frac{1}{2}$$
$$= \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \sum_{n=0}^{\infty} \left(\frac{1}{2} \cdot \frac{1}{2}\right)^n$$
$$= \frac{1}{8} \cdot \sum_{n=0}^{\infty} \left(\frac{1}{4}\right)^n = \frac{1}{8} \cdot \frac{1}{1-\frac{1}{4}} = \frac{1}{8} \cdot \frac{1}{\frac{3}{4}} = \frac{1}{8} \cdot \frac{4}{3} = \frac{1}{6}$$

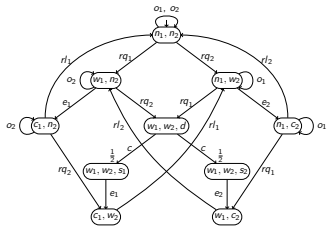
Probabilistic models: Markov decision processes

A *Markov decision process* (MDP) is a tuple $\mathcal{M} = (S, \bar{s}, L, Act, \mathrm{P})$ where

- S is a finite set of states
- $\bar{s}$ is the initial state
- $L\colon S \to \Sigma$ is a labelling function
- Act is a finite set of actions
- $\mathrm{P}\colon S \times Act \times S \to [0,1]$ is the transition probability matrix

P is such that $\sum_{s' \in S} \mathrm{P}(s, a, s') \in \{0, 1\}$ for each $s \in S$ and $a \in Act$.

Example of Markov decision process

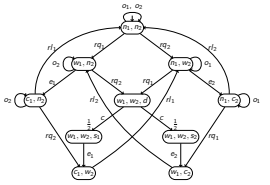


$\mathcal{M} = (S, \bar{s}, L, Act, \mathrm{P})$

- $S = \{(n_1, n_2), (n_1, w_2), \ldots, (w_1, w_2, d), (w_1, w_2, s_1), (w_1, w_2, s_2), \ldots\}$
- $\bar{s} = (n_1, n_2)$
- $L(s) = s$ for each $s \in S$
- $Act = \{o_1, o_2, rq_1, rq_2, e_1, e_2, c, \ldots\}$
- $\mathrm{P}((n_1, n_2), o_1, (n_1, n_2)) = 1$
  $\mathrm{P}((n_1, n_2), e_1, (n_1, n_2)) = 0$
- $\mathrm{P}((w_1, w_2, d), c, (w_1, w_2, s_1)) = \frac{1}{2}$
  $\mathrm{P}((w_1, w_2, d), c, (w_1, w_2, s_2)) = \frac{1}{2}$
  $\mathrm{P}((w_1, w_2, d), c, (w_1, w_2, d)) = 0$
  ...

Computing probabilities

What is the probability of finally reaching the state (c_1, w_2)?



$\mathfrak{P}((n_1, n_2)\,(w_1, n_2)\,(w_1, w_2, d)\,(w_1, w_2, s_1)\,(\overline{(c_1, w_2)})\,)$
$+ \mathfrak{P}((n_1, n_2)\,(w_1, n_2)\,(c_1, n_2)\,(\overline{(c_1, w_2)})\,)$
$+ \mathfrak{P}((n_1, n_2)\,(n_1, n_2)\,(w_1, n_2)\,(c_1, n_2)\,(\overline{(c_1, w_2)})\,)$
$+ \mathfrak{P}((n_1, n_2)\,(w_1, n_2)\,(w_1, n_2)\,(c_1, n_2)\,(\overline{(c_1, w_2)})\,)$
$+ \mathfrak{P}((n_1, n_2)\,(w_1, n_2)\,(c_1, n_2)\,(c_1, n_2)\,(\overline{(c_1, w_2)})\,)$
$+ \mathfrak{P}((n_1, n_2)\,(w_1, n_2)\,(w_1, n_2)\,(c_1, n_2)\,(c_1, n_2)\,(\overline{(c_1, w_2)})\,)$
$+ \mathfrak{P}((n_1, n_2)\,(w_1, n_2)\,(c_1, n_2)\,(n_1, n_2)\,(w_1, n_2)\,(w_1, w_2, d)\,(w_1, w_2, s_1)\,(\overline{(c_1, w_2)})\,)$
...

Computing probabilities

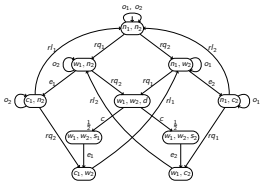
How can we compute $\mathfrak{P}((n_1, n_2)\,(w_1, n_2)\,(c_1, n_2)\,\textcircled{(c_1, w_2)})$?
Intuitively, it is

$$\mathfrak{P}((n_1, n_2)\,(w_1, n_2)\,(c_1, n_2)\,\textcircled{(c_1, w_2)})$$
$$= \mathrm{P}((n_1, n_2), rq_1, (w_1, n_2)) \cdot \mathrm{P}((w_1, n_2), e_1, (c_1, n_2)) \cdot \mathrm{P}((c_1, n_2), rq_2, \textcircled{(c_1, w_2)})$$

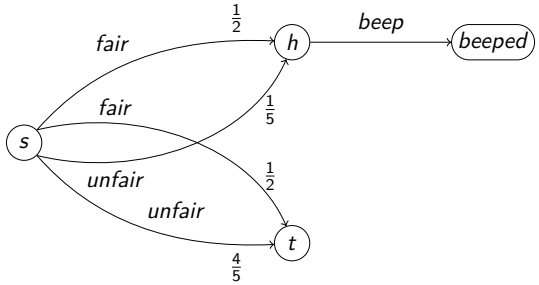
And what about $\mathfrak{P}((n_1, n_2)\,(n_1, n_2)\,(w_1, n_2)\,(c_1, n_2)\,\textcircled{(c_1, w_2)})$?
Intuitively, it is

$$\mathfrak{P}((n_1, n_2)\,(w_1, n_2)\,(c_1, n_2)\,\textcircled{(c_1, w_2)})$$
$$= \mathrm{P}((n_1, n_2), o_2, (n_1, n_2)) \cdot \mathrm{P}((n_1, n_2), rq_1, (w_1, n_2))$$
$$\cdot\, \mathrm{P}((w_1, n_2), e_1, (c_1, n_2)) \cdot \mathrm{P}((c_1, n_2), rq_2, \textcircled{(c_1, w_2)})$$

So, what do we do in (n_1, n_2)?



Computing probability, a simplified example



What is the probability of having $\textcircled{beeped}$?

Computing probability by resolving nondeterminism

In order to compute the probability of a path, we need a means to resolve nondeterminism.
Nondeterminism is resolved by a *scheduler* (policy, adversary, strategy, ...) on a given path.
Paths now include the performed actions.

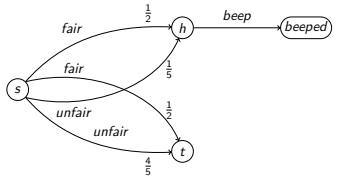
Scheduler

Given an MDP $\mathcal{M}$, a scheduler is a function $\upsilon\colon \mathit{Paths}^*(\mathcal{M}) \to \mathit{Dist}(\mathit{Act})$ such that whenever $\upsilon(\xi)(a) > 0$, then $\sum_{s' \in S} \mathrm{P}(\mathit{last}(\xi), a, s') = 1$.

The probability of a finite path $\xi = s_0\, a_1\, s_1\, a_2\, \ldots\, s_n$ is then defined recursively as:

$$\mathfrak{P}(s_0 a_1 s_1 a_2 \ldots s_n) = \begin{cases} 1 & \text{if } \xi = \bar{s}, \\ 0 & \text{if } \xi = s_0 \neq \bar{s}, \\ \mathfrak{P}(\xi') \cdot \upsilon(\xi')(a_n) \cdot \mathrm{P}(\mathit{last}(\xi'), a_n, s_n) & \text{if } \xi = \xi'\, a_n\, s_n. \end{cases}$$

Computing probability, a simplified example



What is the probability of having $\textcircled{beeped}$?
It is $\mathfrak{P}(s\ \mathit{fair}\ h\ \mathit{beep}\ \textcircled{beeped}) + \mathfrak{P}(s\ \mathit{unfair}\ h\ \mathit{beep}\ \textcircled{beeped})$.

For the scheduler υ choosing *fair* in s, it is $\frac{1}{2}$.
For the scheduler υ choosing *unfair* in s, it is $\frac{1}{5}$.
For the scheduler υ choosing uniformly between *fair* and *unfair* in s, it is $\frac{7}{20}$.

The probabilistic branching time logic PCTL

The logic PCTL expresses properties about the branching structure of the system. Examples:

- with probability 1, an IP address is eventually obtained
- two processes are in the critical section at the same time with probability 0
- if a process wants to enter the critical section, with probability 1 it will eventually enter

Syntax of the PCTL logic

The formal syntax of PCTL is as follows:

$$\varphi ::= a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathbb{P}_{\bowtie p}[\Psi]$$
$$\Psi ::= \mathbf{X}\varphi \mid \varphi\, \mathbf{U}\, \varphi$$

where $a \in AP$ is an atomic proposition, $\bowtie \in \{<, \leq, =, \geq, >\}$, and $p \in [0,1] \cap \mathbb{Q}$.

φ is called a *state* formula while Ψ a *path* formula.

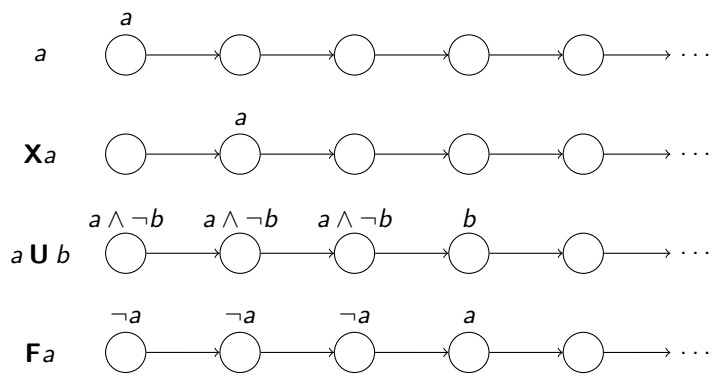
Other common operators can be derived:

$$\mathbf{false} = a \wedge \neg a$$
$$\mathbf{true} = \neg\mathbf{false}$$
$$\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$$
$$\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$$
$$\mathbf{F}\varphi = \mathbf{true}\, \mathbf{U}\, \varphi$$

Examples of PCTL formulas

- with probability 1, an IP address is eventually obtained
  $\mathbb{P}_{=1}[\mathbf{F}\mathit{IP}]$
- two processes are in the critical section at the same time with probability 0
  $\mathbb{P}_{=0}[\mathbf{F}(c_1 \wedge c_2)]$
- if a process wants to enter the critical section, with probability 1 it will eventually enter
  $\bigwedge_{i=1}^{2}(w_i \rightarrow \mathbb{P}_{=1}[\mathbf{F}c_i])$

Informal semantics of the PCTL logic



$\mathbb{P}_{\bowtie p}[\Psi]$ if $\mathfrak{P}(\{\, \xi \in \mathit{Paths} \mid \xi \models \Psi \,\}) \bowtie p$

The probabilistic linear temporal logic PLTL

The logic LTL expresses properties about sequences of events.
Examples:

- an IP address is eventually obtained
- it never happens that two processes are in the critical section at the same time
- whenever a process wants to enter the critical section, it will eventually enter

The probabilistic extension PLTL considers the probability of such sequences of events.

Syntax of the PLTL logic

The formal syntax of PLTL is as follows:

$$\begin{aligned} \varphi ::= a \mid \varphi \wedge \varphi \mid \neg\varphi \\ \mid \mathbf{X}\varphi \mid \varphi \,\mathbf{U}\, \varphi \\ \mathrm{PLTL} ::= \mathbb{P}_{\bowtie p}[\varphi] \end{aligned}$$

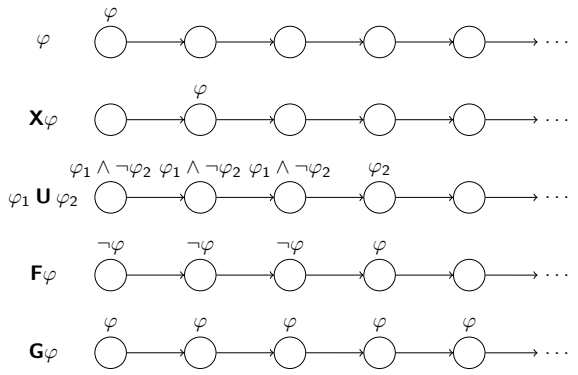
where $a \in AP$ is an atomic proposition, $\bowtie \in \{<, \leq, =, \geq, >\}$, and $p \in [0,1] \cap \mathbb{Q}$.
Other common operators can be derived:

$$\begin{aligned} \mathbf{false} &= a \wedge \neg a \\ \mathbf{true} &= \neg\mathbf{false} \\ \varphi_1 \vee \varphi_2 &= \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ \varphi_1 \rightarrow \varphi_2 &= \neg\varphi_1 \vee \varphi_2 \\ \mathbf{F}\varphi &= \mathbf{true}\,\mathbf{U}\,\varphi \\ \mathbf{G}\varphi &= \neg\mathbf{F}\neg\varphi \end{aligned}$$

Examples of PLTL formulas

- an unused IP address is eventually obtained with probability at most 0.01
  $\mathbb{P}_{\leq 0.01}[\mathbf{F}\mathit{unusedIP}]$
- with probability 1, it never happens that two processes are in the critical section at the same time
  $\mathbb{P}_{=1}[\mathbf{G}\neg(c_1 \wedge c_2)]$
- with probability at least 0.99, whenever a process wants to enter the critical section, it will eventually enter
  $\mathbb{P}_{\geq 0.99}[\mathbf{G}(\bigwedge_{i=1}^{2} w_i \rightarrow \mathbf{F}c_i)]$

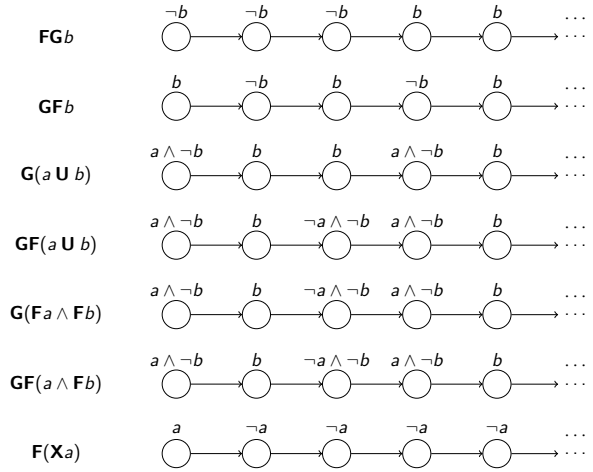
Informal semantics of the PLTL logic



$\mathbb{P}_{\bowtie p}[\varphi]$ if $\mathfrak{P}(\{\,\xi \in \mathit{Paths} \mid \xi \models \varphi\,\}) \bowtie p$

Informal semantics of the PLTL logic: some example

Are these properties satisfied by the following paths?



Mixing PCTL and PLTL: PCTL*

- PCTL is a branching time logic used for stating properties on the branching structure of a system
- PLTL is a linear time logic used for stating properties on the temporal behaviour of a system

How can we state properties on both branching and temporal aspects of a system?

Just use a combination of PCTL and PLTL: PCTL*.

Syntax of the PCTL* logic

The formal syntax of PCTL* is as follows:

$$\varphi ::= a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathbb{P}_{\bowtie p}[\Psi]$$
$$\Psi ::= \varphi \mid \Psi \wedge \Psi \mid \mathbf{X}\Psi \mid \Psi\ \mathbf{U}\ \Psi$$

where $a \in AP$ is an atomic proposition, $\bowtie \in \{<, \leq, =, \geq, >\}$, and $p \in [0,1] \cap \mathbb{Q}$.

φ is called a *state* formula while Ψ a *path* formula.

Other common operators can be derived:

$$\mathbf{false} = a \wedge \neg a$$
$$\mathbf{true} = \neg\mathbf{false}$$
$$\psi_1 \vee \psi_2 = \neg(\neg\psi_1 \wedge \neg\psi_2)$$
$$\psi_1 \rightarrow \psi_2 = \neg\psi_1 \vee \psi_2$$
$$\mathbf{F}\Psi = \mathbf{true}\ \mathbf{U}\ \Psi$$
$$\mathbf{G}\Psi = \neg\mathbf{F}\neg\Psi$$

Examples of PCTL* formulas

- With probability 1, a state which is followed by an error state with probability at most 0.01 is reached infinitely often
  $\mathbb{P}_{=1}[\mathbf{GF}\mathbb{P}_{\leq 0.01}[\mathbf{X}\mathit{error}]]$
- with probability 1, it is never the case that with probability greater than 0.1 an error state is eventually reached
  $\mathbb{P}_{=1}[\mathbf{G}\neg\mathbb{P}_{\geq 0.1}[\mathbf{F}\mathit{error}]]$
- with probability at least 0.99, whenever a process wants to enter the critical section, it will enter within 4 steps with probability at most 0.25
  $\mathbb{P}_{\geq 0.99}[\mathbf{G}(\bigwedge_{i=1}^{2} w_i \rightarrow \mathbb{P}_{\leq 0.25}[c_i \vee \mathbf{X}(c_i \vee \mathbf{X}(c_i \vee \mathbf{X}(c_i \vee \mathbf{X}c_i))))])]$

Reward decorated systems

- Markov chains and Markov decision processes represent probabilistic systems
- We can evaluate the probability of events and check PCTL* formulas
- We have no way to analyze the average cost/reward of getting an event

To overcome the last problem, Markov chains and Markov decision processes can be enriched with reward structures

(State-based) reward structure

Given a Markov chain $\mathcal{M}$, a (state-based) reward structure is a function $\mathrm{rew}\colon S \to \mathbb{N}$.

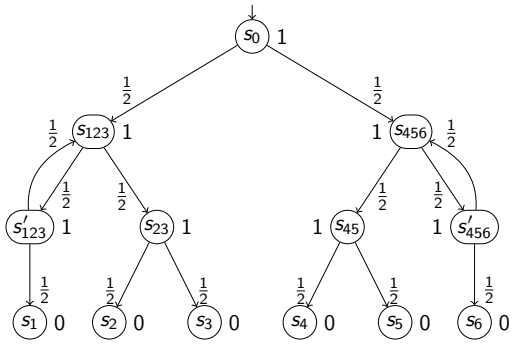
For Markov decision processes can have transition-based reward structures

(Transition-based) reward structure

Given a Markov decision process $\mathcal{M}$, a (transition-based) reward structure is a function $\mathrm{rew}\colon S \times \mathit{Act} \to \mathbb{N}$.

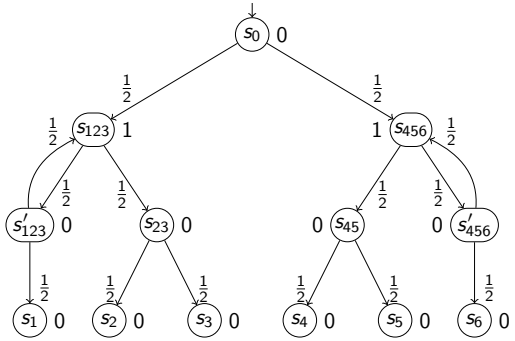
Examples of reward decorated systems

Count the number of coin flips needed to get an outcome



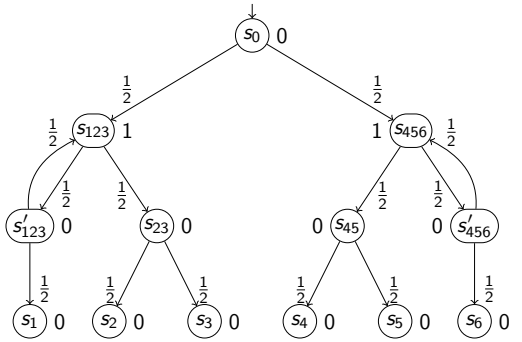
Examples of reward decorated systems

Count the number of rounds needed to get an outcome



Example of path reward

Reward of the path $\xi = s_0\, s_{123}\, s'_{123}\, s_{123}\, s_{23}\, s_2$



Intuitively, it is

$$\begin{aligned}\mathrm{rew}(\xi) &= \mathrm{rew}(s_0) + \mathrm{rew}(s_{123}) + \mathrm{rew}(s'_{123}) + \mathrm{rew}(s_{123}) + \mathrm{rew}(s_{23}) + \mathrm{rew}(s_2)\\ &= 0 + 1 + 0 + 1 + 0 + 0\\ &= 2\end{aligned}$$

Computing rewards

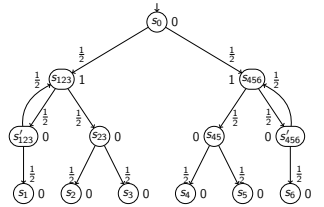
Given a Markov chain $\mathcal{M}$ and a reward structure rew, it is possible to compute[1]:

- the accumulated reward on a path $\xi = s_0\, s_1\, \ldots\, s_n$
  $\mathrm{rew}(\xi) = \sum_{i=0}^{n} \mathrm{rew}(s_i)$
- the δ-discounted accumulated reward on a path $\xi = s_0\, s_1\, \ldots\, s_n$
  $\mathrm{rew}_\delta(\xi) = \sum_{i=0}^{n} \delta^i \cdot \mathrm{rew}(s_i)$
- the expected reward:
  $\mathit{ExpRew} = \sum_{\xi \in \mathit{Paths}} \mathfrak{P}(\xi) \cdot \mathrm{rew}(\xi)$
- the δ-discounted expected reward:
  $\mathit{ExpRew}_\delta = \sum_{\xi \in \mathit{Paths}} \mathfrak{P}(\xi) \cdot \mathrm{rew}_\delta(\xi)$

[1] Some assumption is omitted here for simplicity

Example of accumulated reward

Expected number of rounds needed to get an outcome



From the definition, it is $\mathit{ExpRew} = \sum_{\xi \in \mathit{Paths}} \mathfrak{P}(\xi) \cdot \mathrm{rew}(\xi)$.
In practice, it is

$$\mathit{ExpRew} = 6 \cdot \sum_{r=1}^{\infty} \frac{1}{2} \cdot \left(\frac{1}{2} \cdot \frac{1}{2}\right)^{r-1} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot r = 6 \cdot \frac{1}{8} \cdot \sum_{r=1}^{\infty} r \cdot \left(\frac{1}{4}\right)^{r-1} = \frac{3}{4} \cdot 4 \cdot \sum_{r=1}^{\infty} r \cdot \left(\frac{1}{4}\right)^{r}$$

$$\overset{\dagger}{=} \frac{3}{4} \cdot 4 \cdot \frac{\frac{1}{4}}{(1-\frac{1}{4})^2} = \frac{3}{4} \cdot \frac{1}{\left(\frac{3}{4}\right)^2} = \frac{1}{\frac{3}{4}} = \frac{4}{3}$$

† by the formula $\sum_{k=1}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}$

Interval Markov chains

- Markov chains and Markov decision processes represent probabilistic systems
- We can evaluate the probability of events and check PCTL* formulas
- Sometimes, exact probabilities are unknown

To overcome the last problem, Markov chains and Markov decision processes can be defined using intervals of probabilities instead of precise probabilities

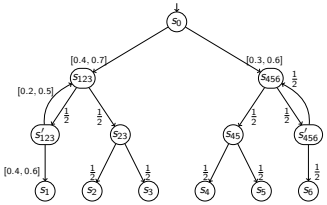
Interval Markov chain

An *interval Markov chain* (IMC) is a tuple $\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$ where

- S is a finite set of states
- $\bar{s}$ is the initial state
- $L\colon S \to \Sigma$ is a labelling function
- $\mathrm{P}\colon S \times S \to \mathbb{I}$ is the interval transition matrix

where $\mathbb{I} = \{[a, b] \subseteq [0, 1]\}$

Example of Interval Markov chain



$\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$

- $S = \{s_0, s_{123}, s_{456}, s'_{123}, s_{23}, s_{45}, s'_{456}, s_1, s_2, s_3, s_4, s_5, s_6\}$
- $\bar{s} = s_0$
- $L(s) = s$ for each $s \in S$
- $\mathrm{P}(s_0, s_0) = [0, 0]$
  $\mathrm{P}(s_0, s_{123}) = [0.4, 0.7]$
  $\mathrm{P}(s_0, s_{456}) = [0.3, 0.6]$
  ...

Interval Markov decision processes

- Markov chains and Markov decision processes represent probabilistic systems
- We can evaluate the probability of events and check PCTL* formulas
- Sometimes, exact probabilities are unknown

To overcome the last problem, Markov chains and Markov decision processes can be defined using intervals of probabilities instead of precise probabilities

Interval Markov decision process

An *interval Markov decision process* (IMDP) is a tuple $\mathcal{M} = (S, \bar{s}, L, Act, \mathrm{P})$ where

- S is a finite set of states
- $\bar{s}$ is the initial state
- $L\colon S \to \Sigma$ is a labelling function
- Act is a finite set of actions
- $\mathrm{P}\colon S \times Act \times S \to \mathbb{I}$ is the interval transition matrix

Parametric Markov chains

- Markov chains and Markov decision processes represent probabilistic systems
- We can evaluate the probability of events and check PCTL* formulas
- Sometimes, exact probabilities are unknown, not even the interval they belong to

To overcome the last problem, Markov chains and Markov decision processes can be defined using functions that are parametric

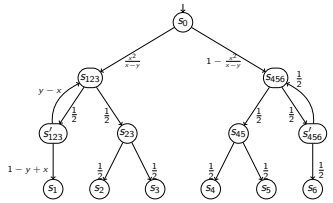
Parametric Markov chain

A *parametric Markov chain* (PMC) is a tuple $\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$ where

- S is a finite set of states
- $\bar{s}$ is the initial state
- $L\colon S \to \Sigma$ is a labelling function
- $\mathrm{P}\colon S \times S \to \mathit{PolyFrac}(\mathbf{P})$ is the parametric transition matrix

where $\mathit{PolyFrac}(\mathbf{P})$ is the set of polynomial fractions over the set of parameters $\mathbf{P}$

Example of Parametric Markov chain



$\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$

- $S = \{s_0, s_{123}, s_{456}, s'_{123}, s_{23}, s_{45}, s'_{456}, s_1, s_2, s_3, s_4, s_5, s_6\}$
- $\bar{s} = s_0$
- $L(s) = s$ for each $s \in S$
- $\mathrm{P}(s_0, s_0) = [0, 0]$
  $\mathrm{P}(s_0, s_{123}) = \frac{x^2}{x-y}$
  $\mathrm{P}(s_0, s_{456}) = 1 - \frac{x^2}{x-y}$
  $\ldots$

Parametric Markov chains

- Markov chains and Markov decision processes represent probabilistic systems
- We can evaluate the probability of events and check PCTL* formulas
- Sometimes, exact probabilities are unknown, not even the interval they belong to

To overcome the last problem, Markov chains and Markov decision processes can be defined using functions that are parametric

Parametric Markov decision process

A *parametric Markov decision process* (PMDP) is a tuple $\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$ where

- S is a finite set of states
- $\bar{s}$ is the initial state
- $L\colon S \to \Sigma$ is a labelling function
- Act is a finite set of actions
- $\mathrm{P}\colon S \times Act \times S \to \mathit{PolyFrac}(\mathbf{P})$ is the parametric transition matrix

The Modelling Language

- IscasMC supports two different modeling languages
- JANI: JSON-based; humans-readable (kind of)
  but intended to be automatically processed
- PRISM modeling language: human readable
- will go into details for JANI
- then short description of PRISM and comparison

JANI - JSON Automata Network Interface

- joint effort between Twente, ISCAS, Aachen, and Argentina
- intended for automatic processing
- in principle human readable
  but not intended to be done so (mainly for debugging)
- two parts:
  model description language
  tool interaction specification
- intended to be easy to parse and write
- intended to be easily extensible by new features
- based on JSON

JSON - JavaScript Object Notation

- human-readable format which stores data as value-object pairs
- definition EBNF

$$
\begin{aligned}
\texttt{Json} &::= \texttt{Number} \mid \texttt{String} \mid \texttt{Boolean} \mid \texttt{Array} \mid \texttt{Object} \mid \texttt{null} \\
\texttt{Number} &::= \text{integer or real number} \\
\texttt{String} &::= \text{`''`} \text{ text } \text{`''`} \\
\texttt{Boolean} &::= \texttt{true} \mid \texttt{false} \\
\texttt{Array} &::= \text{`[` `]`} \mid \text{`[`} \texttt{Json}(\texttt{,}\ \texttt{Json})^* \text{`]`} \\
\texttt{Object} &::= \{(\texttt{String} : \texttt{Json})^*\}
\end{aligned}
$$

- example

```
{"isAlive": true,
  "age": 25,
  "phoneNumbers": [
    { "type": "home", "number": "212 555-1234" },
    { "type": "office", "number": "646 555-4567"}
  ],
  "spouse": null}
```

- document types can be described in `js-schema`
  `https://github.com/molnarg/js-schema`

JANI model description

- JANI model descriptions consist of (incomplete list)
- model type (`dtmc`, `ctmc`, `mdp`, ...)
- list of actions
- list of global variables
- initial values description
- automata description
  local variables
  initial values description system behaviour
- system composition description
- properties description

js-scheme JANI model description

```
var Model = schema({
  "jani-version": Number.min(1).step(1),
  "name": String,
  "?metadata": Metadata,
  "type": ModelType,
  "?features": Array.of(ModelFeature),
  "?actions": Array.of({
    "name": Identifier,
    "?comment": String
  }),
  "?constants": Array.of(ConstantDeclaration),
  "?variables": Array.of(VariableDeclaration),
  "?restrict-initial": {
    "exp": Expression,
    "?comment": String
  },
  "automata": Array.of(Automaton),
  "system": Composition
  "?properties": Array.of(Property),
});
```


JANI version

- `jani-version` is an integer version number
- currently 1
- initial tool paper will be published with this version
- afterwards, will hopefully receive external feedback
- and then start working on version 2

Model name

- `name` is the name of the model
- not interpreted usually
- thus arbitrary

Metadata

- `metadata` contain further information about the model

```
var Metadata = schema({
  "?version": String,
  "?author": String,
  "?description": String,
  "?doi": String,
  "?url": String,
});
```

- `version`: information about the version of this model (e.g. the date when it was last modified)
- `author`: information about the creator of the model
- `description`: a description of the model
- `doi`: the DOI of the paper where this model was introduced/used/described
- `url`: a URL pointing to more information about the model

Model type

- type specifies the semantics type of the model
- list of official model types below
- note: obviously, not all supporting JANI support all of them
- inofficial types can be specified using a name prefixed with "x-"

```
var ModelType = schema([
  "lts",   // labelled transition system
           // (Kripke structure, finite state automaton)
  "dtmc",  // discrete-time Markov chain
  "ctmc",  // continuous-time Markov chain
  "mdp",   // discrete-time Markov decision process
  "ctmdp", // continuous-time Markov decision process
  "ma",    // Markov automaton
  "ta",    // timed automaton
  "pta",   // probabilistic timed automaton
  "sta",   // stochastic timed automaton
  "ha",    // hybrid automaton
  "pha",   // probabilistic hybrid automaton
  "sha"    // stochastic hybrid automaton
]);
```


Additional model features

- features are extensions which are not bound to a particular model type

```
var ModelFeature = schema([
  "derived-operators",
  "nondet-selection",
  "arrays",
  "datatypes",
  "functions",
  "trigonometric-functions",
  "hyperbolic-functions"
]);
```

- further extensions to be specified
- feature names starting with "x-" will not be defined and are available for internal use

Actions

- are used to synchronise several parts of a model
- have a name and an informal description

```
  "?actions": Array.of({
    "name": Identifier,
    "?comment": String
  }),
```


Constants

- constant values do not change over time
- used to define values such as
  number of processes
  concrete probability values
  initial number of molecules
- may be left open to be specified externally
- have a certain type (e.g. integer, real, etc.)

```
var ConstantDeclaration = schema({
  "name": Identifier, // unique constant's name
  "type": [ BasicType, BoundedType ], // the constant's type
  "?value": ConstantValue, // the constant's value, of type type
  "?comment": String // optional comment
});
```

Global variables

- values can change over time
- readable/writable by all automata of the model
- value of non-`transient` variables preserved if not changed

```
var VariableDeclaration = schema({
  "name": Identifier, // unique variable's name
  "type": Type, // the variable's type
  "?transient": [ true, false ], // transient variable if present and
      true
  "?initial-value": [ // unrestricted if not present
    null, // the default value of type
    Expression // a constant expression of type type
  ],
  "?comment": String // an optional comment
});
```


Initial values

- restricts initial values of global variables
- compared to `initial-value`, more complex restrictions possible
- in particular, depending on several variables
  e.g. $x = 1 \vee y = 2$

```
"?restrict-initial": {
  "exp": Expression,
  "?comment": String
},
```


Automata specification

- behaviour of JANI files specified by network of *automata*
- these read/write model variables
- contain set of *locations*
- also have their own *local* variables

```
var Automaton = schema({
  "name": Identifier,
  "?variables": Array.of(VariableDeclaration),
  "?restrict-initial": {  "exp": Expression },
  "locations": Locations
  "initial-locations": Array.of(Identifier),
  "edges": Edges
  })
});
```


Locations specification

- automaton always in one given location
- initially, in one of the initial locations
- note: locations are *not* states

```
  Locations Array.of({
    "name": Identifier,
    "?invariant": { "exp": Expression },
    "?transient-values": Array.of({
      "ref": LValue,
      "value": Expression,
    }),
  }),
```

```
  "initial-locations": Array.of(Identifier),
```

Edges specification

- specify changes of modes and variables

```
Edges: Array.of({
  "location": Identifier, // source location
  "?action": Identifier,  // used for synchronisation
  "?rate": {  "exp": Expression  }, // for continuous-tim models
  "?guard": { "exp": Expression }, // when can be executed?
  "destinations": Array.of({ // stochastic choice of locations
    "location": Identifier, // successor locations
    "?probability": { "exp": Expression  }, // branch probability
    "?assignments": Array.of({  // variable assignments of branch
      "ref": LValue, // variable affected
      "value": Expression, // new value
      "?index": Number.step(1),
    }),
  }),
```


Composition specification

- describes how automata interact with each other
- specify *synchronisation vectors*
- favourite synchronisation mechanism of Hubert Garavel
- subsumes CCS, CSP and others
- in `elements`, an automaton can be used multiple times
  a new copy is used each time
- `input-enable`: adds self-loops with given action if needed

```
var Composition = schema({
  "elements": Array.of({
    "automaton": Identifier, // the name of an automaton
    "?input-enable": Array.of(Identifier) // make input enabled
  }),
  "?syncs": Array.of({
    "synchronise": Array.of([ Identifier, null ]),
         // a list of action names or null, same length as elements
    "result": Identifier,
         // an action name, the result of the synchronisation
  }),
});
```


Properties

- properties of the model, e.g. in PCTL/PLTL, usw.
- also contains PRISM-style filters
- do not assume all tools will immediately support all property types specified

```
var Property = schema({
  "name": Identifier, // the unique property's name
  "expression": PropertyExpression // the state-set formula
  "?comment": String // an optional comment
});
var PropertyExpression = schema([
  [...],
  { // until / weak until
    "op": [ "U", "W" ],
    "left": schema.self,
    "right": schema.self,
    "?step-bounds": PropertyInterval,
    "?time-bounds": PropertyInterval,
    "?reward-bounds": Array.of({
      "ref": Expression,
      "accumulate": Array.of([ "steps", "time" ]),
      "bounds": PropertyInterval
    })
  }
]);
```


Semantics

- JANI models represent a model according to its `type` specification
- here, we only consider MDPs
- some constructs depend on model type used
  e.g. `rate` must not be used for discrete-time models
  e.g. clocks are only allows for timed automata variants

Semantics: state space

- consider non-transient *global* variables
- consider automata locations
- consider non-transient *local* variables
  one set for each copy of `automaton` mentioned in the composition
- state: mode of each automata plus value assignment for all variables
  e.g. one automaton with modes m1, m2, local variable boolean z
  global variables $\{x, y\}$, x integer, y boolean
  valid states:
  $(m1, x = 2, y = \mathit{false}, z = \mathit{false})$,
  $(m1, x = 1, y = \mathit{false}, z = \mathit{true})$,
  $(m2, x = -7, y = \mathit{true}, z = \mathit{false})$,
  ...

Initial states

- initial modes possible depend on `initial-modes`
- for variable, potentially use `initial-value` of variable, if given
- complex expressions possible using `restrict-initial`
- initial states: states fulfilling conjunction of restrictions
  e.g. one automaton
  no local variables, global variables x, y
  singleton set of initial nodes $m1$
  `initial-value` of x is 2
  `restrict-initial` states $x = 2 \rightarrow y = \mathit{false}$
  then the only initial state is $(m1, x = 2, y = \mathit{false})$

Labeling function

- labels each state with value of non-transient global variables
- transient global variables visible using `transient-values` of locations
- in properties, use expressions over states
  e.g. $(m1, x = 2, y = \mathit{false}, z = \mathit{false}) \models x > 1 \wedge \neg z$
  $\mathbb{P}_{\max=?}(\mathbf{F} \models x > 1 \wedge \neg z)$

Actions

- *Act* derived from `actions` definition plus one "invisible" action

```
"?actions": Array.of({
  "name": Identifier,
  "?comment": String
}),
```

Transitions - single automaton

- consider a given state s
- an edge of a given automaton is *enabled* if `location` agrees and `guard` fulfilled
- for single automaton, each `destination` chosen by given `probability`
- for given `destination`, `assignments` change state variables leading to an according successor state

```
Edges: Array.of({
  "location": Identifier, // source location
  "?action": Identifier,  // used for synchronisation
  "?rate": {  "exp": Expression  }, // for continuous-tim models
  "?guard": { "exp": Expression }, // when can be executed?
  "destinations": Array.of({ // stochastic choice of locations
    "location": Identifier, // successor locations
    "?probability": { "exp": Expression  }, // branch probability
    "?assignments": Array.of({  // variable assignments of branch
      "ref": LValue, // variable affected
      "value": Expression, // new value
      "?index": Number.step(1)
    }),
  })
```


Transitions - synchronisation

- consider state s
- for each `automaton` of `elements` consider all enabled edges
- the `composition` specifies the *synchronisation vector*

```
var Composition = schema({
  "elements": Array.of({ "automaton": Identifier }),
  "?syncs": Array.of({
    "synchronise": Array.of([ Identifier, null ])
    "result": Identifier
  })
});
```

e.g.

Automaton 1	Automaton 2	Automaton 3	Result
a	a	a	a
b!	b?	`null`	τ

- each row states possible synchronisation
- e.g. first row: all three automata perform *a*-labelled edge
- e.g. second row: automaton 1 and 2 perform b! resp. b?; 3 not involved
- nondeterministic choice between sets of edges executed together

Transitions - executing edges

- effects of all non-`null` edges executed together
- probabilities of destinations multiplied
- e.g. assume have $\{e1, e2, e3\}$ with
  $destinations(e1) = [d1 \mapsto 0.5, d2 \mapsto 0.5]$
  $destinations(e2) = [d3 \mapsto 0.3, d4 \mapsto 0.4, d5 \mapsto 0.3]$
  $destinations(e3) = [d6 \mapsto 0.25, d7 \mapsto 0.75]$
- then combined destinations are
  $destinations(e1 : e2 : e3) = [$
  $d1 : d3 : d6 \mapsto 0.5 \cdot 0.3 \cdot 0.25,$
  $d1 : d3 : d7 \mapsto 0.5 \cdot 0.3 \cdot 0.75,$
  $\ldots$
  $]$
- all assignments of destinations executed together
- thus, synchronised edges *must not* write write to same variable
- (except if using `index` feature, not discussed here)

PRISM

- PRISM: most widely used probabilistic model checker
- input language: extension of Dijkstra's guarded commands
- similar semantics as in JANI

```
module two_chains
  m : [0..3];
  x : int;

  [a] m=0 -> 1.0: (x'=1000) & (m'=1);
  [b] m=0 -> 1.0: (x'=2) & (m'=1);
  [c] m=1 & x>0 -> 0.3: (x'=x-1) + 0.7: (m'=3);
  [d] m=1 & x<=0 -> 1.0: (m'=2);
endmodule

init
  m = 0 & x = 0
endinit
```

Comparison JANI-PRISM

JANI	PRISM
JSON-based	text-based
hardly readable	human-readable
easily extensible	extensions not that easy
used in IscasMC, Modest toolchain, Prophesy, ...	used in PRISM, Ymer, etc.
synchronisation vectors	CSP-style synchronisation
locations and guards	guarded commands
lts, dtmc, ctmc, mdp, ctmdp, ma, ta, pta, sta, ha, pha, sha	dtmc, ctmc, mdp, pta (+smg)
preparing initial tool paper	been around for quite a while

JANI interaction

- JANI specification also contains *JANI interaction*
- client/server architecture
- e.g. "analyse these properties of this model"
  "the value of this property is as follows: ..."
- based on JSON and standard I/O streams or WebSockets
- `https://docs.google.com/document/d/1RMIkSd99FP3rfwTITXIXZIeVQHoBBjPhiw6QSVTqfxU/edit?ts=567a86f7#`

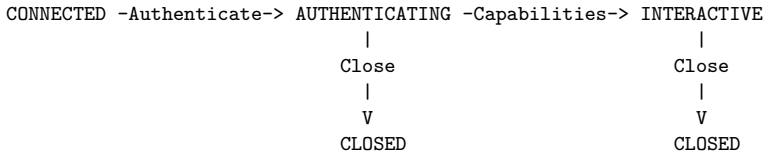
JANI input/output: standard streams

- UTF-8 without byte order mark
- one message (JSON document) per line
- useable if client/server run on same machine
- e.g. model checking competition:
  referee client starts according model checking servers, starts tasks

JANI input/output: WebSockets

- client and server can be on different machines
- usable e.g. for web interface
- uses WebSocket text messages containing JSON documents
- port 15291 (unencrypted) / 15292 (secure WebSocket) by default

Protocol state machine



Authentification

- client sends authentification to server
- contains list of JANI versions supported by client
- optional list of JANI interaction extensions supported by client
- optional login data

```
var Authenticate = schema({
  "jani-versions": Array.of([ Number.min(1).step(1) ]),
  // jani-interaction versions supported by the client
  "?extensions": Array.of(Extension), // extensions supported
  "?login": String,
  "?password": String
});
```


Capabilities

- sent by server as answer of Authentification message
- `type`: type of the message
- `jani-version`: JANI interaction version used from now on
- `extensions`: extensions supported by both
- `metadata`: server metadata
- `parameters`: e.g. BDD engine to use
- `roles`: e.g. model checking, model transformation, etc.

```
var Capabilities = schema({
  "type": "capabilities",
  "jani-version": Number.min(1).step(1),
  "?extensions": Array.of(Extension),
  "metadata": Metadata,
  "?parameters": Array.of(ParameterDefinition),
  "roles": Array.of([
    "analyse",
    "transform"
  ]),
});
```


Close connection

- sent by client or server to terminate a connection

```
var Close = schema({
  "type": "close",
  "?reason": String
});
```

Setting parameters

- client to server
- `type`: identifies message type
- set parameters such as BDD engine etc.
- `id`: used to identify message instance

```
var RequestUpdateServerParameters = schema({
  "type": "server-parameters",
  "id": Number.min(1).step(1)
  "parameters": Array.of(ParameterValue)
});
```


Answer setting parameters

- server to client
- `type`: identifies message type
- answers setting parameter request
- `id`: identifies message instance answered by this reply
- `error`: error message, e.g. "invalid parameter combination", etc.

```
var ReplyUpdateServerParameters = schema({
  "type": "server-parameters",
  "id": Number.min(1).step(1),
  "success": [ true, false ],
  "?error": String
});
```


Start analysis

- client to server
- `type`: identifies message type
- `id`: used to identify message instance
- `engine`: explicit, BDD, etc.
- `modelling-formalism`: used to support non-JANI models
- `model`: model to analyse
- `properties`: identifies properties of the model to analyse
- `parameters`: analysis-specific parameter settings
- `experiments`: allows to run a set of analyses at once

```
var StartAnalysisTask = schema({
  "type": "analysis-start",
  "id": Number.min(1).step(1),
  "engine": Identifier,
  "?modelling-formalism": Identifier,
  "model": AnyModel,
  "?properties": Array.of(Identifier),
  "parameters": Array.of(ParameterValue),
  "?experiments": Array.of({
    "experiment": Identifier,
    "values": Array.of({
      "name": Identifier,
      "value": [ true, false, Number ]
    })
  })
});
```


Provide analysis results

- server to client
- `type`: identifies message type
- `id`: identifies analysis request message answered
- `results`: contains all or some of the results of analysis request

```
var ProvideAnalysisResults = schema({
  "type": "analysis-results",
  "id": Number.min(1).step(1),
  "results": AnalysisDataSet
});
```

IscasMC

- mainly written in Java, with some parts written in C
- modular approach to perform model checking
- targets at using proven techniques from software engineering
  e.g. appropriate use of patterns (builder, delegate, etc.)
- targets at appropriate documentation using JavaDoc
- divided into core parts and plugins (division not completely fixed)
- uses Maven, Ant, and make for the build process
- Eclipse should be developed for development
- uses external component where appropriate

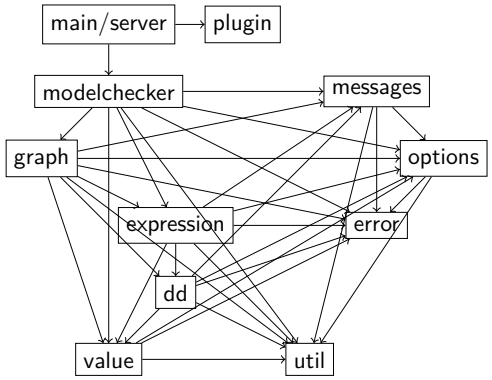
External components used

- in core part:
  - Google Guava
  - Trove
  - Java Native Access
  - JavaCC
  - Java JSON library
- additionally in one or more plugins:
  - SPOT (LTL model checking)
  - LPSolve (multi-objective model checking, IMDP lumping)
  - BDD libraries (BeeDeeDee, BuDDy, CacBDD, CUDD, JDD, Meddly, Sylvan)
    BDD-based symbolic model checking
  - iSat3, other SMT solvers by SMTLIB interface (parametric model checking)
  - Java Algebra System/CoCoA, GiNaC (parametric model checking)
  - GMP/MPFR (arbitrary-precision model checking; experimental)
  - SQLite (JANI interaction)
- trying to keep number of dependencies low
- new dependencies should be encapsuled in according plugin

Core part

Core part divided into components
Division between core part and plugins not yet completely fixed,
some of them might be transformed to plugin

- `util`
- `main/server`
- `modelchecker`
- `graph`
- `expression`
- `value`
- `options`
- `messages`
- `error`
- `dd`
- `plugin`



`util` component

- contains classes and static methods not found in Google Guava
- utility classes
  - Stop watches
  - bit sets (more flexible ones)
  - bit-valued stack
  - integer deque
  - ordered maps/sets
  - bit-stream classes
  - ...
- static methods
  - instantiation helpers
  - get manifest entries
  - print stack trace
  - resource to string
  - ...

main/server components

- sets up model checking process
- can start model checking process in new process
  to prevent bugs crash the whole program
  (important for web-based usage)

modelchecker component

- `ModelChecker` coordinates model checking process
- model checking uses certain `Engines`
  - `EngineExplicit`
  - `EngineDD`
  - ...
- maps expression types to according solvers for given engine
- activates properties necessary for checking given properties
  e.g. according reward structures
- `Model` interface
  - represents models of given type, e.g. PRISM, JANI, etc.
  - load model from file
  - parse properties
  - add external properties
  - create low-level model for given engine

graph component

- representation of graphs for multiple purposes
  e.g. for low-level model or for automata
- explicit-state and BDD-based representations
- interfaces for graphs, node- and edge properties
- predefined graphs e.g. using sparse matrix structure
- predefined node and edge property classes
- state-space explorer to turn high-level model to graph
- conversion from BDD-based graphs to explicit-state graphs
- other auxiliary classes

expression component

- maintaining expressions, such as $(a = 2) \wedge \neg(b = c)$, $\text{PMAX}{=}?(\text{F}(a\text{U}b))$
- expressions are immutable
- used both for models (e.g. on guards) and properties
- contains basic expression types
  e.g. application of operators (e.g. "+", "−", "∧", etc.
- but less general expression types can be added by plugins
- fast evaluation of expressions
- simplification of expressions
- transformation to BDDs

value component

- type system
- Type objects used to generate new Value
  e.g. TypeDouble.newValue() creates new ValueDouble
- allows for typing expressions (see last slide)
- expression literals contain a single immutable Value
- for performance, Values are *modifiable*
- Operator classes to combine values
  particularly used by expression package
- Value may implement methods for addition, subtraction, etc.
- simple replacement of e.g. probability types allows to support
  - arbitrary/infinite precision computations
  - interval Markov chains or IMDPs
  - parametric models
  - quantum Markov chains
  - ...

  while reusing most parts of existing code
- for performance, specialisation is necessary
  e.g. evaluate expressions using Java doubles

options component

- management of options, e.g.
  engine to use, BDD package to use
- "trivial", but important component
- quite a lot of options now
- must integrate with plugin system
- due to huge number of options, hierarchical view necessary
- blocks invalid values for options
- descriptions are stored in resource files
  - eases correction of spelling mistakes etc.
  - improves code readibility
  - eases later internationalisation

messages component

- manages output of information to users
- message: identifier plus arguments
- user readable message text stored in resource files
- better than direcly using System.out
- non-translated output easier to parse by other tools
- allows later internationalisation
- allows sending messages through a network (cf. JANI interaction)

error component

- manages problems
- uses single exception type IscasMCException (inspired by PRISM)
- identifier to get information about specific problem
  e.g. parsing error, probabilities larger than one, etc.
- can be annotated by position information
  (problem caused by construct in which file, line, column?)
- user readable description stored in resource file
- non-translated output easier to parse by other tools
- allows later internationalisation
- allows sending errors through a network (cf. JANI interaction)

dd component

- management of decision diagrams like BDDs and MTBDDs
- relies on external BDD packages, each specified by a plugin
  e.g. BeeDeeDee, BuDDy, CacBDD, CUDD, JDD, Meddly, Sylvan
- wrapper accessing these packages in a uniform way
- supports working with BDD representations of different variable types
- supports symbolically applying `Operators` on such BDDs
  for translating expressions to (MT)BDDs
- auxiliary functions not present in all packages
  e.g. enumeration of satisfying assignments
  arbitrary *n*-ary operations
- conversion of BDDs to Java memory
  faster in some cases

plugin component

- responsible for loading plugins
- can either be JAR files or directories
- defined interfaces to call functions at specific points of time (“hooks”)
- after a command has been executed
- before a model has been parsed
- after a model has been parsed
- after program options have been created

Plugins

- loaded by `plugin` component
- serve a number of different purposes
- commands: `check`, `explore`, `expression2automaton`, `help`, `lump`
- BDD packages: `beedeedee`, `buddy`, `cacbdd`, `cudd`, `jdd`, `meddly`, `sylvan`
- property solvers: `propositional`, `pctl`, `ltl-lazy`, `filter`, `reward`, `coalition`, `multiobjective`
- automata: `automata`, `automata-schewe`
- constraint solving: `lp-solve`, `isat3`, `smt-lib`
- graph-based solvers: `graphsolver`, `graphsolver-iterative`, `graphsolver-lp`
- high-level model description languages: `jani-model`, `prism`, `rddl`
- special semantical model types: `qmc`, `imdp`, `param`, `timedautomata`
- GUI support: `jani-interactions`
- high/arbitrary precision model checking: `mpfr`, `gmp`
- hiding not strictly necessary options for tool evaluation: `specialise-qmc`, `specialise-smg`

Property solver plugins

- available solver classes are stored in a field of the `Options`
- using hook, property solver plugins add according class after options creation
- given expression to be checked
  `ModelChecker` creates instance of available solvers for given `Engine`
  calls `bool canHandle()` of this instance
  if `true`, calls `solve()`

Other plugin types

- lists of candidate classes also used for other means, e.g.
- `Operators`
- commands
- graph solvers
- high-level model types
- constraint solvers
- message types of JANI interaction plugin

Build process for distribution

- distinction between building for development and building for distribution because requirements are very different
- for distribution, use Maven plus Ant and some shell scripts
- packs and optimises IscasMC into one JAR
- can also add required plugins
- also supports building multi-platform JAR files using cross compilers
- build time rather high, but time not that relevant

Development using Eclipse

- for development, Eclipse should be used
- chosen because most widespread JAVA IDE, and one of the best
- build for development uses internal Eclipse building tools
- does *not* rely on Maven
- multi-platform support not needed for this task
- build time very important, to avoid annoyance during programming
- note: make sure all plugin projects are opened as well such that changes (renaming etc.) are propagated to all of them

Versioning

- currently using Subversion server located at ISCAS `http://124.16.137.63/svn/iscasmc/tool`
- planning to make tool open source, probably GPL
- will most likely use Github then

PCTL model checking problem

PCTL model checking problem

Input: a finite MC $\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$, a state $s \in S$, and a (state) PCTL formula φ

Output: yes, if $s \models \varphi$; no, otherwise

Before presenting the algorithm for solving the PCTL model checking problem, we need to formalize the meaning of $s \models \varphi$.

Probabilistic models: Markov chains

A *(Discrete time) Markov chain* (MC) is a tuple $\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$ where

- S is a finite set of states
- $\bar{s}$ is the initial state
- $L\colon S \to \Sigma$ is a labelling function
- $\mathrm{P}\colon S \times S \to [0,1]$ is the transition probability matrix

P is such that $\sum_{s' \in S} \mathrm{P}(s, s') \in \{0, 1\}$ for each $s \in S$.

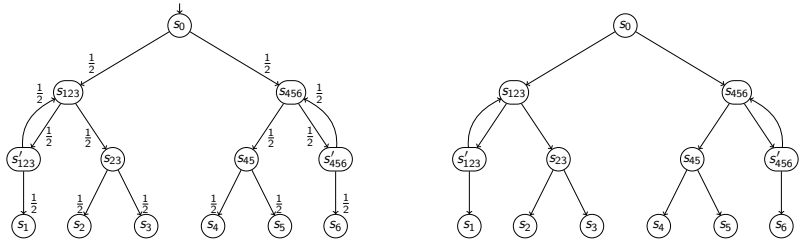
Graph underlying a MC

Every Markov chain induces an underlying directed graph corresponding to the possible transitions

Underlying graph

Given a MC $\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$, the underlying directed graph of $\mathcal{M}$ is the graph $\mathcal{G}(\mathcal{M}) = (V, E)$ where

- $V = S$ is the set of *vertices* and
- $E = \{\, (s, s') \in V^2 \mid \mathrm{P}(s, s') > 0 \,\}$ is the set of *edges*/*arcs*.



Paths of a MC

Given a MC $\mathcal{M}$, a path ξ in $\mathcal{M}$ is a (possibly infinite) sequence of states $\xi = s_0\, s_1\, s_2 \ldots$ such that for each $i \geq 0$, $\mathrm{P}(s_i, s_{i+1}) > 0$.

Alternatively, a path ξ in $\mathcal{M}$ is a (possibly infinite) sequence of states $\xi = s_0\, s_1\, s_2 \ldots$ such that ξ is a path in the underlying graph $\mathcal{G}(\mathcal{M})$.

We denote by $\mathit{Paths}(\mathcal{M})$ the set of all infinite paths of $\mathcal{M}$, and by $\mathit{Paths}^*(\mathcal{M})$ the set of all finite paths of $\mathcal{M}$.

Syntax of the PCTL logic

The formal syntax of PCTL is as follows:

$$\varphi ::= a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathbb{P}_{\bowtie p}[\Psi]$$
$$\Psi ::= \mathbf{X}\varphi \mid \varphi \,\mathbf{U}\, \varphi$$

where $a \in AP$ is an atomic proposition, $\bowtie \in \{<, \leq, =, \geq, >\}$, and $p \in [0,1] \cap \mathbb{Q}$.

φ is called a *state* formula while Ψ a *path* formula.

Other common operators can be derived:

$$\mathbf{false} = a \wedge \neg a$$
$$\mathbf{true} = \neg\mathbf{false}$$
$$\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$$
$$\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$$
$$\mathbf{F}\varphi = \mathbf{true}\,\mathbf{U}\,\varphi$$

Formal semantics of PCTL formulas

Given a MC $\mathcal{M}$, a state $s \in S$, and a path $\xi = s_0\, s_1\, s_2\, \ldots \in \mathit{Paths}(\mathcal{M})$, let $\xi[i]$ denote the state s_i; the formal semantics of PCTL formulas is as follows:

Satisfaction relation for state formulas

$s \models a$	iff	$a \in L(s)$
$s \models \neg\varphi$	iff	it is not the case that $s \models \varphi$
$s \models \varphi_1 \wedge \varphi_2$	iff	$s \models \varphi_1$ and $s \models \varphi_2$
$s \models \mathbb{P}_{\bowtie p}[\Psi]$	iff	$\mathfrak{P}_s(\{\, \xi \in \mathit{Paths}(\mathcal{M}) \mid \xi \models \Psi \,\}) \bowtie p$

Satisfaction relation for path formulas

$\xi \models \mathbf{X}\varphi$	iff	$\xi[1] \models \varphi$
$\xi \models \varphi_1 \,\mathbf{U}\, \varphi_2$	iff	there exists $i \in \mathbb{N}$ such that $\xi[i] \models \varphi_2$ and for each $0 \leq j < i$, $\xi[j] \models \varphi_1$

Satisfaction relation for PCTL formulas

Given a MC $\mathcal{M}$ and a (state) PCTL formula φ, $\mathcal{M} \models \varphi$ if $\bar{s} \models \varphi$.

Formal treatment of the probabilistic operator $\mathbb{P}$

The semantics of $\mathbb{P}_{\bowtie p}[\Psi]$ is defined by comparing $\mathfrak{P}_s(\{\, \xi \in \mathit{Paths}(\mathcal{M}) \mid \xi \models \Psi \,\})$ with p.

How is $\mathfrak{P}_s(\{\, \xi \in \mathit{Paths}(\mathcal{M}) \mid \xi \models \Psi \,\})$ formally defined?

Some recall from measure theory is needed.

Measurability theory: sample space

Sample space

A *sample space* Ω of an experiment is a set of elements that have a 1-to-1 relationship with the possible outcomes of the experiment.

Examples of sample spaces:

- $\Omega = \{T, H\}$ for the outcome of flipping a coin
- $\Omega = \{1, 2, 3, 4, 5, 6\}$ for the outcome of tossing a coin
- $\Omega = \{O, E\}$ for the parity of the outcome of tossing a coin
- $\Omega = \{0, 1, \ldots, 36\}$ for the outcome of spinning a roulette

Measurability theory: σ-algebra

σ-algebra

For a sample space $\Omega \neq \emptyset$, a *σ-algebra* is a pair $(\Omega, \mathcal{F})$; $\mathcal{F} \subseteq 2^\Omega$ is a collection of subsets of the sample space such that:

- $\Omega \in \mathcal{F}$
- for each $A \in \mathcal{F}$, it is $(\Omega \setminus A) \in \mathcal{F}$ — completement
- for a set $\{ A_i \in \mathcal{F} \mid i \in \mathbb{N} \}$, it is $\left(\bigcup_{i \in \mathbb{N}} A_i\right) \in \mathcal{F}$ — countable union

The elements in $\mathcal{F}$ of a σ-algebra $(\Omega, \mathcal{F})$ are called *events*.
The pair $(\Omega, \mathcal{F})$ is called a *measurable set*.

Examples of measurable spaces:

- for any $\Omega \neq \emptyset$, $(\Omega, \{\emptyset, \Omega\})$ is the smallest σ-algebra
- for any $\Omega \neq \emptyset$, $(\Omega, 2^\Omega)$ is the largest σ-algebra
- for $\Omega = \{1,2,3,4,5,6\}$, $\mathcal{F} = \{\emptyset, \{1,3,5\}, \{2,4,6\}, \Omega\}$ defines the σ-algebra $(\Omega, \mathcal{F})$ where the parity is measured

Probability space

Probability space

A *probability space* is a structure $(\Omega, \mathcal{F}, \mathfrak{P})$ such that:

- $(\Omega, \mathcal{F})$ is a σ-algebra
- $\mathfrak{P} \colon \mathcal{F} \to \mathbb{R}^{\geq 0}$ is a *probability measure*, that is:
  - $\mathfrak{P}(\Omega) = 1$
  - $\mathfrak{P}\left(\bigcup_{i \in I} A_i\right) = \sum_{i \in I} \mathfrak{P}(A_i)$ for any $A_i \in \mathcal{F}$ such that $A_i \cap A_j = \emptyset$ for each $i, j \in I$, $i \neq j$

Examples of probability spaces:

- $(\{T,H\}, \{\emptyset, \{T\}, \{H\}, \{T,H\}\}, \mathfrak{P})$ with $\mathfrak{P}(\emptyset) = 0$, $\mathfrak{P}(\{T\}) = \mathfrak{P}(\{H\}) = \frac{1}{2}$, and $\mathfrak{P}(\{T,H\}) = 1$ is the probability space of a fair coin
- $(\{T,H\}, \{\emptyset, \{T\}, \{H\}, \{T,H\}\}, \mathfrak{P})$ with $\mathfrak{P}(\emptyset) = 0$, $\mathfrak{P}(\{T\}) = \frac{3}{4}$, $\mathfrak{P}(\{H\}) = \frac{1}{4}$, and $\mathfrak{P}(\{T,H\}) = 1$ is the probability space of a biased coin
- $(\{1,2,3,4,5,6\}, \{\emptyset, \{1,2,3,4,5,6\}\}, \mathfrak{P})$ with $\mathfrak{P}(\emptyset) = 0$ and $\mathfrak{P}(\{1,2,3,4,5,6\}) = 1$ is the probability space of whether a dice is rolled

Probability space for MCs, intuitively

For model checking PCTL, we are interested in the probability of the paths of the MC.

Intuitively, from a state s, we have:

- Outcomes: the possible paths starting from s
- Events: subsets of these paths
- The paths in each basic event share a common prefix
- The construction is based on cylinder sets

Probability space for MCs, formally

Given a MC $\mathcal{M}$ and a finite path $\hat{\xi} \in \mathit{Paths}^*(\mathcal{M})$, the *cylinder set* $\mathit{Cyl}(\hat{\xi})$ of $\hat{\xi}$ is defined as

$$\mathit{Cyl}(\hat{\xi}) = \{ \xi \in \mathit{Paths}(\mathcal{M}) \mid \hat{\xi} \text{ is a prefix of } \xi \}$$

Probability space of the paths of a MC

The probability space of the paths of a MC $\mathcal{M}$ is the triple $(\Omega, \mathcal{F}, \mathfrak{P})$ where

- $\Omega = \mathit{Paths}^*$
- $\mathcal{F} = \{\emptyset, \Omega\} \cup \{ \mathit{Cyl}(\hat{\xi}) \mid \hat{\xi} \in \mathit{Paths}^*(\mathcal{M}) \}$
- $\mathfrak{P}$ is the unique probability measure defined, for each $\hat{\xi} \in \mathit{Paths}^*(\mathcal{M})$ as:

$$\mathfrak{P}(\mathit{Cyl}(\hat{\xi})) = \begin{cases} 1 & \text{if } \hat{\xi} = s_0 = \bar{s}, \\ 0 & \text{if } \hat{\xi} = s_0 \neq \bar{s}, \text{ and} \\ \mathfrak{P}(\mathit{Cyl}(\hat{\xi}')) \cdot \mathrm{P}(\mathit{last}(\hat{\xi}'), s) & \text{if } \hat{\xi} = \hat{\xi}'\, s \end{cases}$$

If we are interested in the probability of paths of $\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$ starting from a state t, we consider $\mathfrak{P}_t$ defined for the MC $\mathcal{M}_t = (S, t, L, \mathrm{P})$.

Some events of interest in MCs

Given a MC $\mathcal{M}$, we are interested in the following events.

Reachability

Eventually reach a state in the goal set $G \subseteq S$:

$$\mathbf{F}G = \{\, \xi \in \mathit{Paths}(\mathcal{M}) \mid \exists n \in \mathbb{N} : \xi[n] \in G \,\}$$

Invariance

Never leave the goal set $G \subseteq S$:

$$\mathbf{G}G = \{\, \xi \in \mathit{Paths}(\mathcal{M}) \mid \forall n \in \mathbb{N} : \xi[n] \in G \,\} = \overline{\mathbf{F}\overline{G}}$$

Constrained reachability

Eventually reach a state in the goal set $G \subseteq S$ while avoiding bad states $B \subseteq S$:

$$\overline{B}\,\mathbf{U}\,G = \{\, \xi \in \mathit{Paths}(\mathcal{M}) \mid \exists n \in \mathbb{N} : \xi[n] \in G \wedge \forall i < n : \xi[i] \notin B \,\}$$

Some other events of interest in MCs

Given a MC $\mathcal{M}$, we are also interested in the following events.

Repeated reachability

Repeatedly visit a state in the goal set $G \subseteq S$:

$$\mathbf{GF}G = \{\, \xi \in \mathit{Paths}(\mathcal{M}) \mid \forall n \in \mathbb{N} : \exists j \geq n : \xi[j] \in G \,\}$$

Persistence

Eventually reach a state in the goal set $G \subseteq S$ and never leave G afterward:

$$\mathbf{FG}G = \{\, \xi \in \mathit{Paths}(\mathcal{M}) \mid \exists n \in \mathbb{N} : \forall j \geq n : \xi[j] \in G \,\}$$

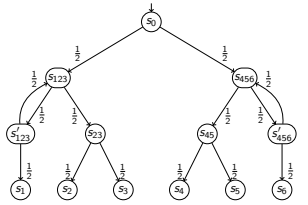
Measurability of events of interest

Measurability of events

Given a MC $\mathcal{M}$ and two sets of states $B, G \subseteq S$, the events $\mathbf{F}G$, $\mathbf{G}G$, $\mathbf{FG}G$, $\mathbf{GF}G$, and $\overline{B}\,\mathbf{U}\,G$ are all measurable.

Computing probabilities

What is the probability of finally reaching the state $\textcircled{s_2}$, i.e., $\mathfrak{P}(\mathbf{F}\textcircled{s_2})$?



$$\begin{aligned}
&\mathfrak{P}(\mathbf{F}\textcircled{s_2}) \\
&= \mathfrak{P}(\{\, \xi \in \mathit{Paths} \mid \exists n \in \mathbb{N} : \xi[n] = \textcircled{s_2} \,\}) \\
&= \mathfrak{P}\Big(\bigcup_{n=0}^{\infty} \mathit{Cyl}(s_0\, s_{123}\, (s'_{123}\, s_{123})^n\, s_{23}\, \textcircled{s_2})\Big) \\
&= \sum_{n=0}^{\infty} \mathfrak{P}(\mathit{Cyl}(s_0\, s_{123}\, (s'_{123}\, s_{123})^n\, s_{23}\, \textcircled{s_2})) \\
&= \sum_{n=0}^{\infty} \frac{1}{2} \cdot \frac{1}{2} \cdot \left(\frac{1}{2} \cdot \frac{1}{2}\right)^n \cdot \frac{1}{2} = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \sum_{n=0}^{\infty} \left(\frac{1}{2} \cdot \frac{1}{2}\right)^n \\
&= \frac{1}{8} \cdot \sum_{n=0}^{\infty} \left(\frac{1}{4}\right)^n = \frac{1}{8} \cdot \frac{1}{1 - \frac{1}{4}} = \frac{1}{8} \cdot \frac{1}{\frac{3}{4}} = \frac{1}{8} \cdot \frac{4}{3} = \frac{1}{6}
\end{aligned}$$

Computing probabilities, a simpler way

Computing probability by means of geometric distributions is:

- not mechanizable
- not always feasible
- error prone

We need to look for a more automatic and reliable approach for computing probabilities.

For reachability properties, this can be achieved by solving a linear equation system

Reachability probabilities in finite MCs

Problem statement

Data: a MC $\mathcal{M}$, a state s, and a set of states $G \subseteq S$

Aim: compute $\mathfrak{P}(s \models \mathbf{F}G) = \mathfrak{P}_s(\mathbf{F}G)$

Remember that $\mathfrak{P}_s(\mathbf{F}G) = \mathfrak{P}_s(\{\, \xi \in \mathit{Paths}(\mathcal{M}) \mid \exists n \in \mathbb{N} : \xi[n] \in G \,\})$

Characterization of reachability probabilities

- For each state $s \in S$, consider the variable x_s representing the probability of satisfying $\mathbf{F}G$ in s, i.e., $x_s = \mathfrak{P}(s \models \mathbf{F}G)$
- For each state $s \in S$,
  - if $s \in G$, then $x_s = 1$
  - if s can not reach G, then $x_s = 0$
  - for each $s \in \mathrm{pred}^*(G) \setminus G$,

$$x_s = \underbrace{\sum_{g \in G} \mathrm{P}(s,g)}_{\text{reach } G \text{ in one step}} + \underbrace{\sum_{t \in S \setminus G} \mathrm{P}(s,t) \cdot x_t}_{\text{reach } G \text{ via } t \in S \setminus G}$$

Predecessor states of a MC

Given a MC $\mathcal{M}$, for $\mathcal{G}(\mathcal{M})$ and a set of vertices $U \subseteq V$, the set of

- immediate predecessors of U is $\mathrm{pred}(U) = \{\, v \in V \mid \exists u \in U : (v,u) \in E \,\}$
- predecessors of U is $\mathrm{pred}^*(U) = \bigcup_{n=0}^{\infty} \mathrm{pred}^n(U)$ where

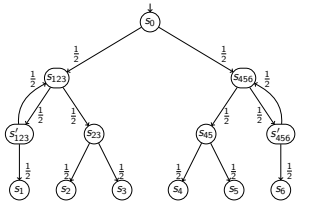
$$\mathrm{pred}^n(U) = \begin{cases} U & \text{if } n = 0 \\ U' \cup \mathrm{pred}(U') & \text{if } n > 0 \text{ with } U' = \mathrm{pred}^{n-1}(U) \end{cases}$$

In practice, pred^* is the reflexive and transitive closure of pred.

Computing probabilities

What is the probability of finally reaching the state $\textcircled{s_2}$, i.e., $\mathfrak{P}(\mathbf{F}\textcircled{s_2})$?
By using the previous construction, we have:

- $x_{s_1} = x_{s_3} = x_{s_4} = x_{s_5} = x_{s_6} = 0$ and $x_{\textcircled{s_2}} = 1$
- $x_{s_{456}} = x_{s'_{456}} = x_{s_{45}} = 0$
- $x_{s_{23}} = \frac{1}{2} \cdot x_{\textcircled{s_2}} + \frac{1}{2} \cdot x_{s_3}$
- $x_{s'_{123}} = \frac{1}{2} \cdot x_{s_{123}} + \frac{1}{2} \cdot x_{s_1}$
- $x_{s_{123}} = \frac{1}{2} \cdot x_{s'_{123}} + \frac{1}{2} \cdot x_{s_{23}}$
- $x_{s_0} = \frac{1}{2} \cdot x_{s_{123}} + \frac{1}{2} \cdot x_{s_{456}}$



By solving the system, we obtain
$x_{s_{23}} = \frac{1}{2}$, $x_{s_{123}} = \frac{1}{3}$, $x_{s'_{123}} = \frac{1}{6}$, and $x_{s_0} = \frac{1}{6}$

Reduced linear equation system

The linear equation system can be reduced by considering only states in $\text{pred}^*(G) \setminus G$:

- let $S_? = \text{pred}^*(G) \setminus G$ be the states not in G that can reach G
- let $\mathbf{A} = (\mathrm{P}(s,t))_{s,t \in S_?}$ be the restriction of P to only states in $S_?$
- let $\mathbf{b} = (b_s)_{s \in S_?}$ be the probability to reach G in 1 step: $b_s = \sum_{g \in G} \mathrm{P}(s,g)$

Then $\mathbf{x} = (x_s)_{s \in S_?}$ with $x_s = \mathfrak{P}(s \models \mathbf{F} G)$ is the *unique* solution of

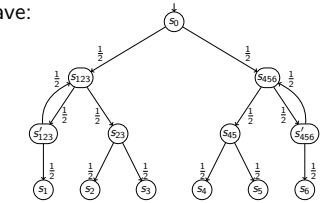
$$\mathbf{x} = \mathbf{A} \times \mathbf{x} + \mathbf{b} \text{ or, equivalently, } (\mathbf{I}_{S_?} - \mathbf{A}) \times \mathbf{x} = \mathbf{b}$$

Computing probabilities

What is the probability of finally reaching the state $\textcircled{s_2}$, i.e., $\mathfrak{P}(\mathbf{F}\textcircled{s_2})$?

By using the previous matrix construction, we have:

- $S_? = \{s_0, s_{123}, s'_{123}, s_{23}\}$
- $$\begin{pmatrix} 1 & -\frac{1}{2} & 0 & 0 \\ 0 & 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & -\frac{1}{2} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} s_0 \\ s_{123} \\ s'_{123} \\ s_{23} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \frac{1}{2} \end{pmatrix}$$



By Gaussian elimination, we obtain
$x_{s_{23}} = \frac{1}{2}$, $x_{s_{123}} = \frac{1}{3}$, $x_{s'_{123}} = \frac{1}{6}$, and $x_{s_0} = \frac{1}{6}$

Improving reachability probability computation

Essentially, we have partitioned the state space into:

- $S_{=1} = G$, the set of states where $\mathbf{F} G$ is trivially satisfied
- $S_{=0} = \{\, s \in S \mid \mathfrak{P}(s \models \mathbf{F} G) = 0 \,\}$, the set of states where $\mathbf{F} G$ never holds
- $S_? = S \setminus (S_{=1} \cup S_{=0})$, the remaining states, where $\mathbf{F} G$ holds with non-zero probability

Actually, we can use any partition satisfying

- $G \subseteq S_{=1} \subseteq \{\, s \in S \mid \mathfrak{P}(s \models \mathbf{F} G) = 1 \,\}$
- $B \setminus G \subseteq S_{=0} \subseteq \{\, s \in S \mid \mathfrak{P}(s \models \mathbf{F} G) = 0 \,\}$
- $S_? = S \setminus (S_{=1} \cup S_{=0})$

In practice, $S_{=1}$ and $S_{=0}$ should be chosen as large as possible, so to have $S_?$ (and the corresponding equation system) as small as possible:

$$S_{=1} = \{\, s \in S \mid \mathfrak{P}(s \models \mathbf{F} G) = 1 \,\} \text{ and } S_{=0} = \{\, s \in S \mid \mathfrak{P}(s \models \mathbf{F} G) = 0 \,\}$$

They can be obtained in linear time by analysing the underlying graph $\mathcal{G}(\mathcal{M})$.

Iterative computation of reachability probabilities

The reduced linear equation system consists of:

- $S_? = \text{pred}^*(G) \setminus G$ being the states not in G that can reach G
- $\mathbf{A} = (\mathrm{P}(s,t))_{s,t \in S_?}$ being the restriction of P to only states in $S_?$
- $\mathbf{b} = (b_s)_{s \in S_?}$ being the probability to reach G in 1 step: $b_s = \sum_{g \in G} \mathrm{P}(s,g)$

Then $\mathbf{x} = (x_s)_{s \in S_?}$ with $x_s = \mathfrak{P}(s \models \mathbf{F} G)$ is the *unique* solution of

$$\mathbf{x} = \mathbf{A} \times \mathbf{x} + \mathbf{b}$$

This gives an iterative approach to compute $\mathbf{x}$:

$$\mathbf{x}^{(0)} = \mathbf{0} \text{ and } \mathbf{x}^{(i+1)} = \mathbf{A} \times \mathbf{x}^{(i)} + \mathbf{b} \text{ for } i \in \mathbb{N}.$$

Then

1. $\mathbf{x}^{(n)}(s) = \mathfrak{P}(s \models \mathbf{F}^{\leq n} G)$ for $s \in S_?$
2. $\mathbf{x}^{(0)} \leq \mathbf{x}^{(1)} \leq \mathbf{x}^{(2)} \leq \ldots \leq \mathbf{x}$
3. $\mathbf{x} = \lim_{n \to \infty} \mathbf{x}^{(n)}$

where $\mathbf{F}^{\leq n} G = \{\, \xi \in \mathit{Paths}(\mathcal{M}) \mid \exists i \leq n : \xi[i] \in G \,\}$ is the bounded version of $\mathbf{F} G$.

Iterative computation of reachability probabilities

The sequence of approximate solutions $\mathbf{x}^{(0)} = \mathbf{0}$ and $\mathbf{x}^{(i+1)} = \mathbf{A} \times \mathbf{x}^{(i)} + \mathbf{b}$ for $i \in \mathbb{N}$ can be obtained by an algorithm computing iteratively $\mathbf{x}^{(i+1)} = \mathbf{A} \times \mathbf{x}^{(i)} + \mathbf{b}$ and stopping when

$$\max_{s \in S_?} |x_s^{(i+1)} - x_s^{(i)}| < \varepsilon \text{ for some small tolerance } \varepsilon$$

This method ensures convergence.

For practical computation, other methods are used, like Gauss-Siedel or Jacobi.

Constrained reachability probabilities in finite MCs

Problem statement

Data: a MC $\mathcal{M}$, a state s, and two sets of states $B, G \subseteq S$

Aim: compute $\mathfrak{P}(s \models \overline{B}\,\mathbf{U}\,G) = \mathfrak{P}_s(\overline{B}\,\mathbf{U}\,G)$

Remember that
$\mathfrak{P}_s(\overline{B}\,\mathbf{U}\,G) = \mathfrak{P}_s(\{\,\xi \in \mathit{Paths}(\mathcal{M}) \mid \exists n \in \mathbb{N} : \xi[n] \in G \wedge \forall i < n : \xi[i] \notin B\,\})$

Characterization of reachability probabilities

- For each state $s \in S$, consider the variable x_s representing the probability of satisfying $\overline{B}\,\mathbf{U}\,G$ in s, i.e., $x_s = \mathfrak{P}(s \models \overline{B}\,\mathbf{U}\,G)$
- For each state $s \in S$,
  - if $s \in G$, then $x_s = 1$
  - if s can not reach G via $\overline{F}$, then $x_s = 0$
  - for each $s \in (\mathrm{pred}^*(G) \cap \overline{F}) \setminus G$,

$$x_s = \sum_{g \in G} \mathrm{P}(s, g) + \sum_{t \in S \setminus G} \mathrm{P}(s, t) \cdot x_t$$

In practice, the same constructions and optimizations for $\mathbf{F}G$ apply.

PCTL model checking problem

We have now all the ingredients for solving the PCTL model checking problem:

PCTL model checking problem

Input: a finite MC $\mathcal{M} = (S, \bar{s}, L, \mathrm{P})$, a state $s \in S$, and a (state) PCTL formula φ

Output: yes, if $s \models \varphi$; no, otherwise

Basic algorithm

The decision about $s \models \varphi$ is made by:

1. Compute the satisfaction set $\mathit{Sat}(\varphi) = \{\, s \in S \mid s \models \varphi \,\}$:
   this is done recursively by a bottom-up traversal of the parse tree of φ:
   - the nodes of the tree are the sub-formulas of φ
   - for each node, i.e., for each subformula η of φ, compute $\mathit{Sat}(\eta)$
   - compute $\mathit{Sat}(\eta)$ by means of the satisfaction sets of its children, like:
     $\mathit{Sat}(\eta_1 \wedge \eta_2) = \mathit{Sat}(\eta_1) \cap \mathit{Sat}(\eta_2)$ and $\mathit{Sat}(\neg\eta) = S \setminus \mathit{Sat}(\eta)$
2. Return $s \in \mathit{Sat}(\varphi)$

Core part of the PCTL model checking algorithm

The satisfaction set $\mathit{Sat}(\,\cdot\,)$ is defined by structural induction as follows:

$$\begin{aligned}
\mathit{Sat}(a) &= \{\, s \in S \mid a \in L(s) \,\} \\
\mathit{Sat}(\varphi_1 \wedge \varphi_2) &= \mathit{Sat}(\varphi_1) \cap \mathit{Sat}(\varphi_2) \\
\mathit{Sat}(\neg\varphi) &= S \setminus \mathit{Sat}(\varphi) \\
\mathit{Sat}(\mathbb{P}_{\bowtie p}[\Psi]) &= \{\, s \in S \mid \mathfrak{P}_s(s \models \Psi) \bowtie p \,\}
\end{aligned}$$

where $s \models \Psi = \{\, \xi \in \mathit{Paths}(\mathcal{M}) \mid \xi \models \Psi \wedge \mathit{fst}(\xi) = s \,\}$.

The **X** next operator case: $\mathfrak{P}(s \models \mathbf{X}\varphi)$

The basic result underlying $\mathfrak{P}(s \models \mathbf{X}\varphi)$ is: $\mathfrak{P}(s \models \mathbf{X}\varphi) = \sum_{s' \in Sat(\varphi)} \mathrm{P}(s, s')$.
This gives the following algorithm:

- consider all states simultaneously
- in matrix representation, it is

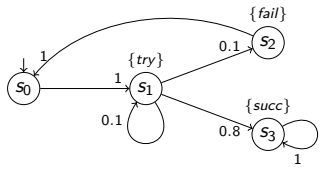
$$(\mathfrak{P}(s \models \mathbf{X}\varphi))_{s \in S} = \mathrm{P} \times \mathbf{b}_\varphi$$

where $\mathbf{b}_\varphi(t) = 1$ if $t \in Sat(\varphi)$, 0 otherwise.

In practice, the next operator case reduces to a matrix-vector multiplication.

Example of the **X** next operator case

Consider the MC on the right and
the PCTL formula $\mathbb{P}_{\geq 0.9}[\mathbf{X}(\neg try \vee succ)]$.



We have that

- $Sat(succ) = \{s_3\}$
- $Sat(try) = \{s_1\}$
- $Sat(\neg try) = S \setminus \{s_1\} = \{s_0, s_2, s_3\}$
- $Sat(\neg try \vee succ) = Sat(\neg try) \cup Sat(succ) = \{s_0, s_2, s_3\}$

We know that $(\mathfrak{P}(s \models \mathbf{X}\varphi))_{s \in S} = \mathrm{P} \times \mathbf{b}_\varphi$ where $\varphi = \neg try \vee succ$.
This yields to:

$$(\mathfrak{P}(s \models \mathbf{X}\varphi))_{s \in S} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0.1 & 0.1 & 0.8 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0.9 \\ 1 \\ 1 \end{pmatrix}$$

Thus, $Sat(\mathbb{P}_{\geq 0.9}[\mathbf{X}(\neg try \vee succ)]) = \{s_1, s_2, s_3\}$.

The **U** until operator case: $\mathfrak{P}(s \models \varphi_1 \mathbf{U}\, \varphi_2)$

The algorithm is as follows:

- compute $S_{=1} = Sat(\mathbb{P}_{=1}(\varphi_1 \mathbf{U}\, \varphi_2))$ by graph analysis
- compute $S_{=0} = Sat(\mathbb{P}_{=0}(\varphi_1 \mathbf{U}\, \varphi_2))$ by graph analysis
- solve the equation system for $S_?$
- assign states to $Sat(\mathbb{P}_{\bowtie p}[\varphi_1 \mathbf{U}\, \varphi_2])$ accordingly

On the importance of pre-computing using graph analysis

1. reduces the number of variables in the linear equation system
2. ensures the uniqueness of the solution of the linear equation system
3. gives exact solutions for states in $S_{=0}$ and $S_{=1}$
4. for qualitative properties, no need to solve the linear equation system

In practice, the until operator case reduces to simple graph analysis and the solution of a linear equation system.

Complexity analysis of the PCTL model checking algorithm

Let $|\varphi|$ be the size of a (state) PCTL formula, i.e., the number of temporal and logical operators in φ.

Complexity theorem

For a finite MC $\mathcal{M}$ and state PCTL formula φ, the PCTL model checking problem can be solved in time

$$\mathcal{O}(Poly(|S|) \cdot |\varphi|)$$

Informal analysis

1. The linear complexity in $|\varphi|$ comes from the need to solve the model checking problem for each node of the parse tree (i.e., computing $Sat(\eta)$ for each subformula η of φ).
2. The worst-case complexity comes from the **U** until operator:
   1. Computing $S_{=0}$ and $S_{=1}$ can be done in linear time.
   2. Solving the equation system on $S_?$ is in $\Theta(|S_?|^3)$.